Dissertation / Doctoral Thesis

Titel der Dissertation / Title of the Doctoral Thesis

Space-Filling Curves for Improved Cache-Locality in Shared Memory Environments

verfasst von / submitted by

Dipl-Ing. Martin Albin Perdacher, BSc

angestrebter akademischer Grad / in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften (Dr. techn.)

Wien, 2020 / Vienna, 2020

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 786 880
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Informatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Inform.Univ. Dr. Claudia Plant
Betreut von / Supervisor:	Prof. Dr. Christian Böhm

Abstract

Today's microprocessors consist of multiple cores each of which can perform multiple additions, multiplications, or other operations simultaneously in one clock cycle. In shared memory environments at least two types of parallelism must be applied to exploit the maximum performance of the algorithm: MIMD (Multiple Instruction Multiple Data) where each core simultaneously perform different operations on different types of input data streams and SIMD (Single Instruction Multiple Data) where within a core, the same operation is executed at once on various data. Additionally, modern microprocessors offer a rich memory hierarchy, including various levels of cache and registers. Some of these memories (like main memory, L3 cache) are big but slow and shared among all cores. Others (registers, cache lines, L1 cache) are fast and exclusively assigned to a single core but small. Only if data access has a high locality, we can avoid excessive data transfers between the different levels of the memory hierarchy. Algorithms in linear algebra are often defined by three or more nested loops. In this thesis, we propose to traverse such loops in an order defined by a space-filling curve, such as the Hilbert or the Morton-order curve. The low-level kernels used in this work are based on Advanced Vector Extensions (AVX), which allow the exploitation of several levels of parallelism in shared memory environments. We apply our space-filling curves in several algorithms ranging from linear algebra (matrix-multiplication, Cholesky decomposition, LU factorization) or clustering (K-means) as well as in database queries (i.e., similarity join).

Kurzfassung

Heutige Mikroprozessoren bestehen aus mehreren Kernen, von denen jeder mehrere Additionen, Multiplikationen oder andere Operationen gleichzeitig in einem Taktzyklus ausführen kann. In Shared Memory Architekturen müssen zumindest zwei Arten von Parallelität angewendet werden, um die maximale Leistung des Algorithmus auszunutzen: MIMD (Multiple Instruction Multiple Data), bei dem jeder Kern gleichzeitig verschiedene Operationen an verschiedenen Typen von Eingangsdatenströmen ausführt, und SIMD (Single Instruction Multiple Data), bei dem innerhalb eines Kerns dieselbe Operation an verschiedenen Daten gleichzeitig ausgeführt wird. Zusätzlich bieten moderne Mikroprozessoren eine reichhaltige Speicherhierarchie mit verschiedenen Ebenen der Caches für jedes der Register. Einige dieser Speicher (wie Hauptspeicher, L3-Cache) sind groß, aber langsam und werden von allen Kernen gemeinsam genutzt. Andere (Register, Cache-Zeilen, L1-Cache) sind schnell und ausschließlich einem einzigen, aber kleinen Kern zugeordnet. Nur wenn die Datenzugriffe eine hohe Lokalität haben, können übermäßige Datentransfer zwischen den Elementen der Speicherhierarchie vermieden werden. Algorithmen in der linearen Algebra werden oft durch drei oder mehreren verschachtelte Schleifen definiert. In dieser Arbeit schlagen wir vor, solche Schleifen in einer Reihenfolge zu durchlaufen, die durch eine raumfüllende Kurve, wie z.B. die Hilbert- oder die Morton-Ordnung definiert ist. Die in dieser Arbeit verwendeten Low-Level-Kernel basieren auf Advanced Vector Extensions (AVX), die die Ausnutzung auf mehreren Ebenen der Parallelität in gemeinsam genutzten Speicherumgebungen ermöglichen. Wir wenden unsere raumfüllenden Kurven in verschiedenen Algorithmen an, die von linearer Algebra (Matrix-Multiplikation, Cholesky-Zerlegung, LU-Faktorisierung) oder Clustering (K-means) bis hin zu Datenbankabfragen (d.h. Similarity Join) reichen.

Acknowledgments

I want to thank both advisors Prof. Claudia Plant and Prof. Christian Böhm, for their continuous support, patience, and encouragement during the past years, which shaped my scientific thinking and work. Their precise and efficient work and their scientific advice contributed to the success of this work. I am also very grateful for the excellent working environment at the University of Vienna.

I would also like to thank my wife Eva and my son Viktor in this way. Both of them often had to cut back in order to give me the necessary time for my doctorate. Evas love, her trust and her unbelievably big heart have given me countless times the peace of mind I need for a doctoral program.

Big thanks also to my collegues and friends Ben, Sahar, Lukas, Lena, Ylli, Theresa, Can, Max, Robert, Katerina, and Ewald. Work would be boring without you.

Bibliographic Note

Most of the results of this thesis were successfully published in high-profile conference proceedings and journal articles. Therefore, the chapters of this thesis are based on the following publications and manuscripts:

- **Chapter 3:** Christian Böhm, Martin Perdacher, and Claudia Plant. 'Multi-core K-means'. In: *Proceedings of the 2017 SIAM International Conference on Data Mining, Houston, Texas, USA, April 27-29, 2017.* Ed. by Nitesh V. Chawla and Wei Wang. SIAM, 2017, pp. 273–281.
  - C. Böhm and M. Perdacher jointly devised the project and the main conceptual ideas and carried out the implementation.
  - M. Perdacher additionally performed experiments, discussed and implemented related work.
  - C. Plant supervised the project and particularly took care for consistency of the claimed contributions, experimentation, and related work.
  - All authors contributed to the development and evaluation of the proposed techniques and to paper writing.
- **Chapter 5 and 6:** The journal version of the paper:

  Christian Böhm, Martin Perdacher, and Claudia Plant. 'A Novel Hilbert Curve for Cache-locality Preserving Loops'. In: *IEEE Transactions on Big Data* (2018), pp. 1–14. ISSN: 2332-7790

  and the previous conference version of this paper:

  Christian Böhm, Martin Perdacher, and Claudia Plant. 'Cache-oblivious loops based on a novel space-filling curve'. In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016.* IEEE Computer Society, 2016, pp. 17–26
  - C. Böhm and M. Perdacher jointly developed the main conceptual ideas, proofs and implementation where C. Böhm was responsible for the characteristics of the proposed extension of the space-filling curve ("FUR-Hilbert") and M. Perdacher developed the integration into various application algorithms like Matrix Multiplication and K-means.
  - C. Plant supervised the project and particularly took care for consistency of the claimed contributions, experimentation, and related work.
  - All authors contributed to the development and evaluation of the proposed techniques and to paper writing.

- **Chapter 7:** Martin Perdacher, Claudia Plant, and Christian Böhm. 'Cache-oblivious High-performance Similarity Join'. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* ACM, 2019, pp. 87–104.
  - M. Perdacher and C. Böhm jointly devised the main conceptual ideas and outlined proofs.
  - M. Perdacher was additionally responsible for the overall algorithm, performed experiments, and carried out implementation.
  - C. Plant supervised the project and particularly took care for consistency of the claimed contributions, experimentation, and related work.
  - All authors contributed to the development and evaluation of the proposed techniques and to paper writing.
- **Chapter 8:** Martin Perdacher, Claudia Plant, and Christian Böhm. 'Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition'. In: *2020 IEEE International Conference on Big Data, BigData 2020, Atlanta, GA, USA, December 10-13, 2020.* Accepted for publication. Dec. 2020.
  - M. Perdacher devised the proposed algorithms, performed experiments and carried out implementation and discussion.
  - C. Plant and C. Böhm supervised the work.
  - All authors contributed to the development and evaluation of the proposed techniques and to paper writing.

All authors agreed on this statement of the responsibilities of the individual authors.

Contents

Abstract i

Kurzfassung iii

Acknowledgments v

Bibliographic Note vii

List of Figures xiii

List of Tables xvii

1 Introduction 1
- 1.1 Contributions 4
- 1.2 Publications 5
- 1.3 Structure of this Thesis 6

2 Parallelism in Shared Memory Environment 9
- 2.1 SIMD and MIMD 9
- 2.2 Parallelism via Instructions 11
- 2.3 Hardware Multithreading 12
- 2.4 General Remarks 13

3 Multi-core K-means 15
- 3.1 Introduction to a *Cache-conscious* Approach 15
- 3.2 K-means 16
- 3.3 Multi-core K-means 16
- 3.4 Cluster ID Coding 22
- 3.5 Experiments 24

3.6 Related Work and Discussion 28
3.7 Conclusion 31

4 Construction of Space-Filling Curves 33
4.1 Introduction and Historical Context 33
4.2 Hilbert Function On the Example of Approximating Polygones 36
4.3 A Quaternary Hilbert Pattern 37
4.4 Observations 39
4.5 L-systems 42
4.6 Concluding Remarks 43

5 Cache-oblivious Hilbert Curve 45
5.1 Introduction 45
5.2 Locality of the Hilbert Curve 46
5.3 Well-known Methods for the Hilbert Curve 46
5.4 Novel Non-recursive Lindenmayer 52
5.5 Nano-Programs 59
5.6 Overall Architecture 62
5.7 Concluding Remarks 63

6 Applications of Cache-oblivious Hilbert Curve 65
6.1 Algorihtms 65
6.2 Experimental Evaluation 69
6.3 Related Work and Discussion 80
6.4 Conclusion 83

7 High-Performance Similarity Join 85
7.1 Introduction 85
7.2 Similarity Join 88
7.3 Preliminaries 92
7.4 The FGF-Hilbert Join 94
7.5 Experimental Evaluation 104
7.6 Related Work and Discussion 118
7.7 Conclusion 121

8 Cache-oblivious Morton-order Curve 123
8.1 Introduction 123

8.2 Generating Morton-order Curves . . . 125
8.3 Parallelization . . . 134
8.4 Application of Morton-order Loops for LU Decompositon and Matrix Multiplication . . . 135
8.5 Experimental Evaluation . . . 139
8.6 Related Work and Discussion . . . 145
8.7 Conclusion . . . 148

9 Energy efficiency on Data Movement 149
9.1 Introduction . . . 149
9.2 Experimental Setup . . . 150
9.3 Experimental Evaluation . . . 151
9.4 Discussion . . . 153
9.5 Conclusion . . . 155

10 Summary and Conclusion 157
10.1 Future work . . . 158

Bibliography 161

A Further Experiments 175
A.1 Runtime Performance on Morton-order Curves . . . 175

List of Figures

1.1 Strategies for space-filling curves. . . . 3

2.1 SISD and SIMD paradigm. . . . 10
2.2 MIMD paradigm. . . . 11

3.1 Loop traversal order over n, k, and d. . . . 17
3.2 Vertical and Horizontal Addition, Permutation of Vectors. . . . 19
3.3 Cluster ID coding in IEEE-754 format. . . . 23
3.4 Comparison to Standard K-means. . . . 26
3.5 Speedup with Varying Number of Threads. . . . 27
3.6 Effect of Cluster-ID Coding. . . . 28
3.7 Scalability of the Different Strategies with n, d, and k. . . . 28

4.1 Iterations of the Hilbert curve. . . . 35
4.2 First three approximations of the Hilbert curve using polygons. . . . 36
4.3 The Hilbert value h in decimal format for the first three iterations of the Hilbert curve. . . . 38
4.4 The Hilbert value h in the quaternary format for the first three iterations of the Hilbert curve. . . . 40

5.1 Comparison of the Traversal Order for Nested Loops (a) and Hilbert Loops (b). An improved locality can be recognized in the histories over time for variable i (c) and j (d), and a considerably improved cache miss rate (e). . . . 47
5.2 Mealy-DFA for Inverse Hilbert: $(i, j) := \mathcal{H}^{-1}(h)$ to generate variables i and j from the Hilbert value h. . . . 48
5.3 Recursive generation of (i, j)-pairs following the Hilbert-curve. . . . 52
5.4 Examples of Nano-programs for Grids Ranging from 2×2 to 4×4 (all having basic orientation $d = 2$). . . . 58
5.5 Placement of 3×2 (red) and 3×3 (green) Grids. . . . 61

5.6 FurHilbertFor $(i,j) \in \{2,...,6\} \times \{0,...,12\}$. 62

6.1 Traversal of Cholesky (a) and Floyd/Warshall (b) 66
6.2 Performance of Matrix Multiplication (Xeon). 69
6.3 Performance of Matrix Multiplication on a Manycore System (Xeon Phi). . . 71
6.4 Experiments on matrix multiplication for a smaller cache size (Laptop). . . . 71
6.5 Cache-hit-rate of the matrix multiplication on L1 cache for different matrix sizes (left) and different thread sizes (right). 72
6.6 Cache-hit-rate of the matrix multiplication on L2 cache for different matrix sizes (left) and different thread sizes (right). 73
6.7 Cache-hit-rate of the matrix multiplication on L3 cache for different matrix sizes (left) and different thread sizes (right). 73
6.8 Cache-hit-rate of the matrix multiplication along the complete cache hierarchy for different matrix sizes (left) and different thread sizes (right). 74
6.9 Experiments on K-means Clustering. 75
6.10 Experiments on Cholesky Decomposition. 77
6.11 Experiments on the Algorithm by Warshall. 78
6.12 Energy efficiency for matrix multiplication. 78
6.13 Energy efficiency for Cholesky decomposition. 79
6.14 Energy efficiency for K-means clustering. 80

7.1 Sorting of 20 vector objects in the Epsilon-Grid Order (EGO). 86
7.2 Imaginary similarity matrix for the ε-similarity join. 87
7.3 Strategies to Process Pairs (i,j) of Objects. 90
7.4 EGO-Join using a FGF-Hilbert loop. 94
7.5 Planning Refinements: Upper bounds of intervals (right side) are stored and condensed; thus larger areas of the (i,j)-space can be efficiently discarded from loop traversal (left). 98
7.6 The gain in performance of EGO and *FGF-Hilbert Join* 107
7.7 Cache misses of Canonical, Hilbert and *FGF-Hilbert Join* 108
7.8 Runtime of each phase in *FGF-Hilbert Join* 109
7.9 (a) Full Uniform. (b) 8 Selective Dimensions and 56 Non-selective Dims. (Uniform, 200K, 64d). 110
7.10 Join with Two Sets. 8 Selective Dimensions and 56 Non-selective Dims. (Uniform, $2 \cdot 200$K, 64d). 111
7.11 Runtime Experiments (Default: Uniformly Distributed, 600K, 8d). 112

7.12 Runtime Experiments on Gaussian Data (Gaussian, 600K, 8d). 113
7.13 Runtime Experiments on Real Data. Properties in Table 7.2 114
7.14 Experiments on Skylake CPU (cf. Fig. 7.9) 115
7.15 Runtime Experiments on Skylake CPU (Default: Uniformly Distributed, 600K, 8d; cf. Figure 7.11). 116
7.16 Speedup experiments on real data. Properties in Table 7.2. 117

8.1 Strategies for space-filling curves, violations of monotonicity properties marked in red. 124
8.2 Morton-order. Interleaving the binary coordinates from i and j yields the binary z-values shown. 126
8.3 И-order. Interleaving the binary coordinates from i and j yields the binary z-values shown. 126
8.4 Z-order traversal (tzcnt). 128
8.5 И-order traversal (tzcnt). 129
8.6 Microcell templates 130
8.7 Microcell placement of 5 $\{2 \times 3\}$, 1 $\{3 \times 3\}$, 5 $\{2 \times 4\}$ and 1 $\{3 \times 4\}$ templates (c.f. Figure 8.6) into a 13x7 grid. 131
8.8 SIMD parallelization. 135
8.9 Comparison of different Morton-order generation approaches (Xeon-Phi). . . . 140
8.10 Matrix-multiplication on Xeon-Phi. 142
8.11 LU decomposition on Xeon-Phi. 143
8.12 Forward and backward substitution. 144
8.13 Results evaluated on Xeon. 145

9.1 Power meter set-up. 151
9.2 Runtime performance of the matrix-multiplication on Xeon-Phi. 152
9.3 Power consumption of the matrix-multiplication on Xeon-Phi. 153
9.4 Energy efficiency of the Matrix-multiplication on Xeon-Phi. 153
9.5 Runtime performance of the LU decomposition on Xeon-Phi. 154
9.6 Power consumption of the LU decomposition on Xeon-Phi. 154
9.7 Energy efficiency of the LU decomposition on Xeon-Phi. 155

10.1 Interesting Space-Filling curves. 159

List of Tables

5.1 Lookup table to derive the direction code d for the odd cases of ℓ. 57
5.2 Lookup table to derive the direction code d for the even cases of ℓ. 58

7.1 Properties of Synthetic Data. 105
7.2 Properties of Real Data. 106

8.1 Code table for processing microcells 129

A.1 Runtime on two different Morton order implementations tested on matrix multiplication. 175
A.2 Runtime on two different Morton order implementations tested on LU decomposition. 176

List of Algorithms

1	Recursive Lindenmayer Algorithm.	51
2	The Non-recursive Lindenmayer Alg.	56
3	Lindenmayer with Nano-programs.	60
4	MORTON-ORDER (pext) loop	127
5	MORTON-ORDER (tzcnt) loop	130
6	Microcell placement with Morton-order (pext). Implemented as a preprocessor macro.	133
7	LU block algorithm	136
8	Canonical LU decomposition	137
9	Backward substitution algorithm	137
10	Forward substitution algorithm	138
11	Matrix multiplication	139

Chapter 1

Introduction

Multi-core processors are the standard microarchitecture of our everyday life. Multiple cores are prevalent in current desktop, workstations, notebooks, smartphones, tablets, NAS systems, or even embedded systems powered by ARM© processors (e.g., Raspberry Pi 4 equipped with 4-core ARM Cortex-A72). Until 2005, single core processors dominated the PC sector. Before that, attempts to increase performance by using two or more single-core processors were rarely made. Instead, the focus was on increasing the clock frequency in addition to new instruction sets such as MMX. Nevertheless, from frequencies of about 4 GHz on, the resulting heat dissipation was no longer manageable. One possibility for further development was the introduction of multi-core processors. The world's first dual-core processor was POWER4, a 1 GHz processor invented by IBM in 2001 [26], and it initiated the transition to multi-core systems. This has been the crucial development responding to the ever increasing demand for computing power [99]. By **Multiple Instruction Multiple Data (MIMD) parallelism**, multi-core systems maximize the amount of data that can be processed while keeping the clock speed and thus the energy consumption manageable.

Besides MIMD, **Single Instruction Multiple Data (SIMD) parallelism** or vectorization has been an important design principle going back to the Cray-1 supercomputer of 1976 [2]. In this parallelization principle, the same instruction is applied to many data streams, as in a vector processor. Modern microprocessors offer instruction sets for efficiently processing single instructions on multiple data. The most common are the instruction sets SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions) and its latest extension to 512-bits AVX-512. For example, even smartphone processors, such as the ARM Cortex-A series of RISC processors, supports with NEON

an instruction set similar to SSE. The instruction sets operate on a reserved set of registers such that SIMD and the usual floating-point unit operations of the CPU core can be interleaved in the same clock cycle. Data within the registers reserved for SIMD (called YMM_0 to YMM_{32} in AVX-512) can be very efficiently manipulated. For example, on the Skylake architecture with AVX-512, we have a vector length of 8 double-precision. Each core is equipped with 2 AVX-512 units, where each unit can perform a Fused-Multiply-Add (FMA) operation, which is an addition and one multiplication at the same time ($dest = (a * b) + c$). Thus, this leads to 32 ($= 8 * 2 * 2$) double precision FLOPs per clock cycle. Compared to the previous Broadwell architecture, the processor speed has doubled, and the cache-hierarchy has significantly grown. However, the L2 cache latency has not improved, and there are only minor improvements in terms of latency for the L1 cache [66]. This known gap between processor speed and memory hierarchies remains, and similar characteristics apply to other common architectures. To scale up data mining methods on current architectures, we need to completely re-engineer algorithms with the opportunities provided by the hardware in mind.

To overcome this speed difference, caches are used to accelerate access to frequently used data. Thus, this demands an additional need to develop software algorithms that consider cache memory. A good part of the available performance cannot be used. The performance achieved by simple algorithms is very often relatively poor. There are various highly optimized libraries such as LAPACK [6], OpenBLAS [138] or the ATLAS project [128], but these implementations are based on hardware-tailored implementations and optimizations or at least automatically tuned for a given hardware, as in ATLAS. These libraries can be considered as **cache conscious algorithms**. They have hardware-related information such as cache-sizes (or the length of the cache lines) as an explicit parameter to exploit the memory hierarchy efficiently. One example for such hard-coded information is the step size in the matrix-matrix multiplication such as:

for $I := 0$ **to** $n-1$ **stepsize** s **do**

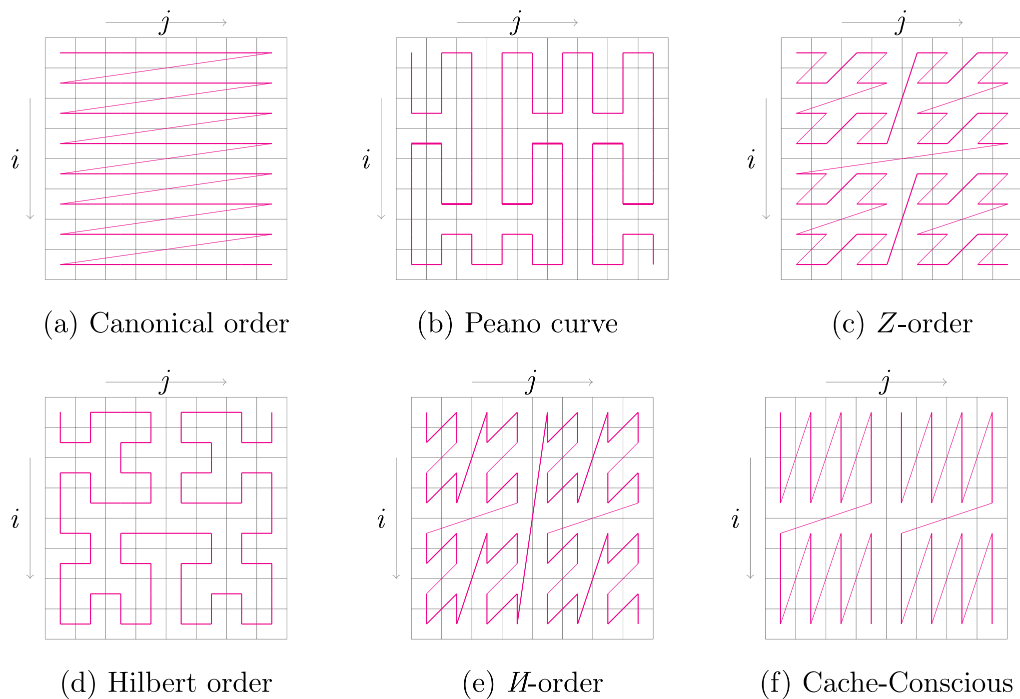
  for $j := 0$ **to** $m-1$ **do**

    for $i := I$ **to** $I{+}s{-}1$ **do** $a_{i,j} := \sum_k b_{i,k} \cdot c^{\mathsf{T}}_{j,k}$

Assuming, that the L1 cache is large enough to store s rows of B and 1 row of C^T. This strategy improves the performance dramatically, because now we have to transfer

C^T from main memory to cache only $\lceil n/s \rceil$ times while we still transfer matrix B once. Cache-conscious approaches have such hard-coded information at multiple loop levels, where the step-sizes are tuned for each hardware and cache level respectively. In this example, we are using the transpose of C, since in C-like languages matrices are stored in a row-wise order and there it is common practice to transpose C before computing the scalar product.

Cache-oblivious algorithms [55] uses the cache optimally ignoring constant factors, such as cache sizes as an explicit parameter. Such an algorithm is designed to perform well on multiple machines with different memory hierarchies without modifications. Thus a cache-oblivious algorithm is designed to work well on multiple machines with different levels of the memory hierarchy of unknown sizes.



(a) Canonical order (b) Peano curve (c) Z-order

(d) Hilbert order (e) $I\!I$-order (f) Cache-Conscious

Figure 1.1: Strategies for space-filling curves.

Typically, a cache-oblivious algorithm works by a recursive divide and conquer strategy. The problem is divided into smaller subproblems, where the size of the subproblem fits into the cache, regardless of its cache size. Instead of this recursive approach, in this thesis we define the size of our problem by a space-filling curve (cf. Figure 1.1), especially the Hilbert curve (1.1d), and the Morton-order curve (Figure 1.1c and e).

There are countless algorithms in various domains, such as graph traversal, clustering, or linear algebra. A nested loop structure is the core of these algorithms. The traversal of such loops over arrays or matrices are typically in canonical order, like Figure 1.1a, or in a block-oriented order such as Figure 1.1f. In this thesis, we propose to replace this nested loop structure with an order defined by a space-filling curve, which has the advantage of preserving the data locality and therefore, supporting today's rich memory hierarchy.

Modern hardware usually supports a memory hierarchy of 2-3 levels where this data locality property enables the benefits of cache-oblivious loops. With the cache-conscious approach, depicted in Figure 1.1f, we outline one prime example of a cache-conscious traversal. Here, we have to emphasize that each block (here 4×4) might depend on the hardware used.

A space-filling curve is a curve in mathematical analysis, whose range contains the entire n-dimensional unit hypercube. In the context of scientific computing and throughout this thesis, we define a one-dimensional ordering of a two-dimensional space, such that each point or cell of the matrix is visited once. The aim is to conserve locality and bridge the gap between processor speed and the speed of memory access.

Space-filling curves have become a valuable tool in many scientific applications where locality in space is essential. The applications in scientific computing are versatile, but all the applications share a common sense of locality. In [111], the authors define a range query based on the Z-order and the Hilbert-curve. The latter is the subject of building up an index for image data [92]. There are also very general approaches of mapping points to space-filling curves [106], but in contrast to these approaches, we exploit the cache hierarchy by replacing the canonical loops with loops, which are defined by a space-filling curve.

1.1 Contributions

In this thesis, we present the following key contributions:

- We present Multi-Core K-means, a **cache-conscious** implementation for today's multi-core microarchitecture.
- We propose to replace nested loops enumerating pairs of (i, j) in canonical order

by **cache-oblivious loops** following a space-filling curve, especially the Hilbert- and the Morton-order curve.

- We overcome the usual limitation of space-filling curves to grids of equal size lengths $n \times n$ where n is a power of 2 or 3.

- We implement our Hilbert- and Morton-order loops as a preprocessor macro, making it extremely convenient to be used as a building block in any host algorithm and facilitates compiler optimization.

- We demonstrate the superiority by applying our loops in several host algorithms, above all the matrix multiplication, which serves here compared to state-of-the-art approaches.

- We introduce the idea of using space-filling curves for the refinement order in a similarity join.

1.2 Publications

Parts of this Ph.D. thesis have been published and presented at international peer-reviewed conferences and journals:

- Christian Böhm, Martin Perdacher, and Claudia Plant. 'Cache-oblivious loops based on a novel space-filling curve'. In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. IEEE Computer Society, 2016, pp. 17–26.

- Christian Böhm, Martin Perdacher, and Claudia Plant. 'Multi-core K-means'. In: *Proceedings of the 2017 SIAM International Conference on Data Mining, Houston, Texas, USA, April 27-29, 2017*. Ed. by Nitesh V. Chawla and Wei Wang. SIAM, 2017, pp. 273–281.

- Christian Böhm, Martin Perdacher, and Claudia Plant. 'A Novel Hilbert Curve for Cache-locality Preserving Loops'. In: *IEEE Transactions on Big Data* (2018), pp. 1–14. ISSN: 2332-7790.

- Martin Perdacher, Claudia Plant, and Christian Böhm. 'Cache-oblivious High-performance Similarity Join'. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 2019, pp. 87–104.

- Martin Perdacher, Claudia Plant, and Christian Böhm. 'Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition'. In: *2020 IEEE International Conference on Big Data, BigData 2020, Atlanta, GA, USA, December 10-13, 2020*. Accepted for publication. Dec. 2020.

1.3 Structure of this Thesis

In this thesis, we focus on the shared memory architecture. The parallelism in such environments is briefly summarized in section 2. We look at MIMD and SIMD parallelization techniques and the memory hierarchy in particular. Chapter 3 we focuses on Multi-core processors and proposes a *cache-conscious* solution (c.f. Figure 1.1f) with today's common cache-hierarchy for the wide-spread clustering algorithm K-means as a highly relevant use-case for knowledge discovery on big data. We propose an entirely re-engineered clustering algorithm focusing on the close connection of the MIMD and SIMD parallelism. However, we observed that the code needs significant changes if we migrate the code to a different hardware setting. Some step-sizes related to the memory hierarchy need fine-tuning to tease out the performance last percentage. We address this issue by rewriting the loops in an order defined by a space-filling curve. Therefore we deal with *cache-oblivious* algorithms in the upcoming chapters. Before diving into space-filling curves and how they are applied in this thesis, we introduce the basic terminology and observations from previous publications in this field in chapter 4.

In chapter 5 we propose to replace nested loops in algorithms with a loop defined by the Hilbert curve (c.f. Figure 1.1d). We revisit well-known methods for generating the

Hilbert curve and propose our non-recursive solution for arbitrarily shaped rectangles. On top of our Hilbert curve, we implemented several algorithms, such as the matrix-multiplication, K-means clustering, Cholesky decomposition, and Floyd's algorithm to find the shortest paths of all pairs in a graph. The implementation details and extensive experiments are summarized in chapter 6.

In chapter 7, we have a detailed look at shared-memory databases, where we apply the Hilbert curve to the ε-similarity join. In particular, we sort the data according to the Epsilon Grid Order (EGO), which serves here as a filter technique. We traverse the remaining part of the non-materialized similarity matrix in a Hilbert-order. In the experiments, we show our technique's various building blocks' impact and demonstrate its use on various synthetic and real-world data sets.

There are use cases where the Hilbert-curve is not applicable because of certain algorithms' data dependencies. LU decomposition is one example. In chapter 8 we introduce our contribution to the Morton-order curve for two different variants, the Z-order and the *И*-order (c.f. Figure 1.1c and e) an inverse version of the Z-order. Extensive experiments on the LU decomposition, matrix-multiplication, and forward- and backward substitution show their superiority over the Hilbert curve.

Space-filling curves like the Hilbert curve or the Morton-order curve reduce the cache misses, and therefore they have an improved data movement pattern. Since data movement is expensive in terms of energy efficiency, it is worth having a look at energy consumption in chapter 9. We measure energy efficiency with an external wattmeter between the power supply and the server to have an undistorted result.

We conclude our thesis in chapter 10 and give some outline on possible future research.

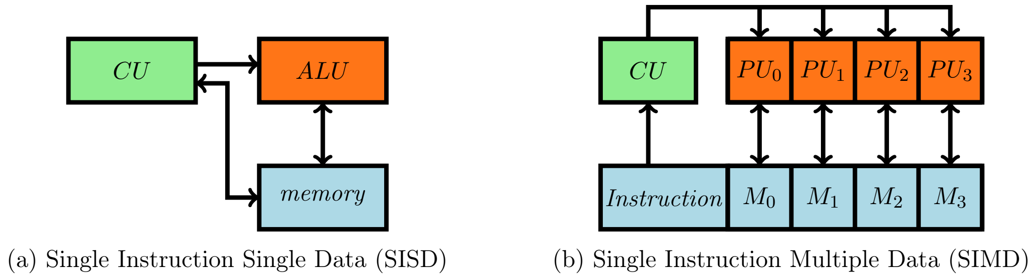
Chapter 2

Parallelism in Shared Memory Environment

In this chapter, we give a high-level introduction to different parallelism levels of shared memory environments. In section 2.1 we introduce into the MIMD and SIMD parallelism, the two most important principles, which are explicitly targeted in our work. In this chapter, we refer to other parallelism techniques such as instruction-level parallelism in section 2.2 or hardware multithreading in section 2.3, but we do not target these techniques directly within this thesis. We conclude this chapter with general remarks in section 2.4.

2.1 SIMD and MIMD

Flynn's taxonomy is a classification of computer architectures proposed in 1966 [53]. This taxonomy has been used as a tool in the design of modern processors and their functionality [48]. Flynn's four classifications are based on the number of concurrent instruction and data streams available in the architecture. The term "stream" refers to a sequence or flow of either instructions or data from the CPU's main memory. The instruction stream is unidirectional, and the data stream is bidirectional. The four-letter abbreviations SISD, SIMD, MISD, and MIMD were derived from the English descriptions' first letters. For example, SISD stands for "Single Instruction stream, Single Data stream". According to Flynn, SISD is a sequential computer with no parallelism in either instruction or data streams, like Intel© Pentium 4. There we have a Control Unit, which takes instructions from memory, which get forwarded to the Arithmetic-Logic Unit (ALU). In Figure 2.1a



(a) Single Instruction Single Data (SISD) (b) Single Instruction Multiple Data (SIMD)

Figure 2.1: SISD and SIMD paradigm.

we visualized the information flow.

All input data are transferred via the ALU's main memory for processing and then written back to memory. Instead of doing this for only one memory cell, this can be done for a whole vector of data. This principle is known as "Single Instruction stream Multiple Data streams" (SIMD) and is nowadays implemented in embedded systems powered by ARM© with the NEON instruction set or in today's laptops as AVX2 or AVX-512 instruction set, c.f., Figure 2.1b. These instructions operate on multiple data elements simultaneously that make the processing of loops more efficient. The programmer does not need to take care of such SIMD instructions because they are generated automatically by the compiler with auto-vectorization. A vectorizing compiler transforms loops into sequences of vector operands. However, if the loop has a complex structure defined by macros, the automatic vectorization fails most of the time. Since we replace canonical loops in algorithms with loops defined by a space-filling curve, it is rarely the case to successfully vectorize such loops. Therefore, we use AVX-512 intrinsics instructions in this thesis directly. These are C style functions that provide access to many instructions, including Intel© SSE, AVX, or AVX-512, without the need to write assembly code. Writing code with such intrinsic instructions simulates the behavior of having an implemented auto-vectorized approach. Nevertheless, we believe that future compilers will profit from the locality assumptions of the Hilbert curve.

The Multiple Instruction stream on Single Data stream (MISD) is an untypical architecture that is generally used for fault tolerance systems, such as a space-shuttle flight control computer [57]. Multiple autonomous processors simultaneously executing different instructions on different data in the Multiple Instruction stream Multiple Data streams (MIMD) paradigm (c.f. Figure 2.2). MIMD architectures include multi-core processors and distributed systems, using either a shared or a distributed memory space.

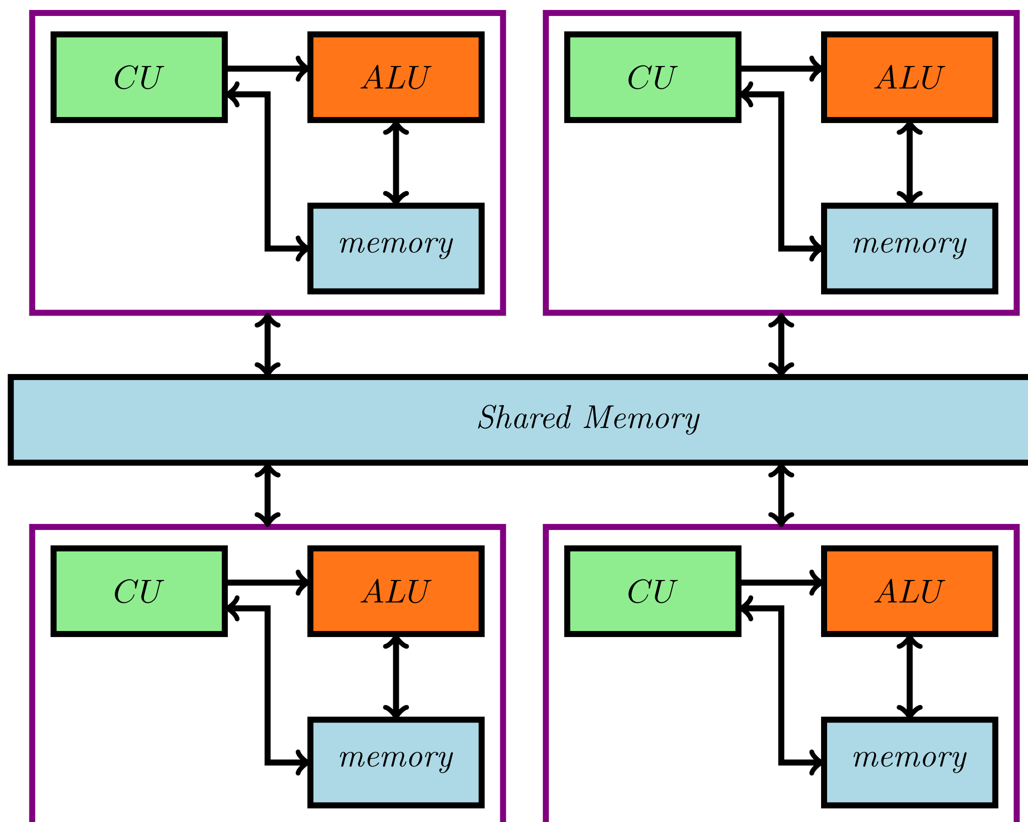


Figure 2.2: MIMD paradigm.

Most parallel computers, as of 2013, are MIMD systems. An example of a MIMD system is the Intel© Xeon Phi™. This processor has multiple processing cores (up to 72 in 2017) to execute different instructions on different data. However, distributed systems with distributed memory environments like a hypercube or mesh interconnection network are also considered MIMD parallel systems. However, throughout the context of this thesis, we focus explicitly on shared memory environments where we refer with the term MIMD parallelism to threading parallelism in multi- or many-core environments. In such multi- and many-core environments, both SIMD and MIMD are combined to achieve maximum performance.

2.2 Parallelism via Instructions

Since about 1985, all processors use pipelining to overlap the execution of instructions and improve the performance. Such a potential overlap among instructions is called

instruction-level parallelism (ILP). As a simple example, considering that reading from memory has a higher latency than performing the actual computation. This effect would cause the CPU to stall. In the meantime, the CPU could decrease a pointer if the mentioned computation does not depend on this pointer. The primary goal is to do some productive work instead of stalling for higher latency operations.

We address this issue by loop unrolling, where multiple copies of the loop body are made. After unrolling, there is more ILP available by overlapping instructions from different iterations. The compiler has then more options to address the ILP.

2.3 Hardware Multithreading

A related concept to MIMD, especially from a programmer perspective, is hardware multithreading. While MIMD relies on multiple processes or threads to keep multiple processors busy, hardware multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion to utilize the hardware resources efficiently. Multithreaded processors, therefore, process several threads quasi-simultaneously. This kind of parallelism can be done in different ways:

- Fine-grained multithreading: A version of hardware multithreading that switches between threads on each instruction. This technique allows each core an interleaved execution of threads. The main goal is to avoid delays or stalls in the instruction stream (=instruction pipeline). These can occur if a thread reads after it writes, and the instruction refers to a result that has not yet been calculated or retrieved.
- Coarse-grained multithreading: This type of multithreading occurs if a thread gets blocked by an event (e.g., L3 cache miss) that typically would create a long latency stall. While this stall might take hundreds of CPU cycles, the threaded processor switches the execution to another thread that is ready to run. The stalled thread is set to ready-to-run after the data arrives.
- Simultaneous multithreading (SMT): Any single thread has a limited amount of ILP. This multithreading type tries to exploit the ILP parallelism across multiple threads to decrease the stall time associated with unused issue slots. This can be done without major changes to the processor architecture: the main additions needed are the ability to fetch instructions from multiple threads and a larger register file to hold data from multiple threads. Often there are four concurrent

threads (or hyper-threads according to the Intel terminology) per CPU core, but some processors support even up to eight concurrent threads per core.

Exploiting these levels of parallelism is far beyond the scope of our thesis. In our experiments, we do not address fine- or coarse-grained multithreading or hyper-threading explicitly. In all of the mentioned cases above, we rely on default compiler optimizations. We test the performance of our algorithms always with one thread per core.

2.4 General Remarks

There are even more opportunities in a shared memory environment to address performance issues explicitly. One example is how processors can access the address space. There are two different styles, the Uniform Memory Access (UMA) and the NonUniform Memory Access (NUMA). In the UMA style, the latency to a word in the memory does not depend on which processor asks for it. In the NUMA style, memory accesses to the High Bandwidth Memory (HBM) are much faster than others, depending on which processor asks which word, typically because the main memory is divided and attached to different microprocessors or different memory controllers on the same chip. However, to take advantage of this architecture, the developer must make the application NUMA-aware. Developers using OpenMP can do so using nested parallelism in OpenMP, where the so-called teams bind to NUMA nodes, and threads in a team use the processors within one NUMA node.

In the case of one of our servers, the Intel© Xeon Phi™, which comes with additional on-package HBM. The HBM can be used as an L3 cache or as a fast addressable memory or in a hybrid model, where half of the memory is used as cache and the other half as fast addressable memory.

Since we want to guarantee fair comparisons, we decided to use the HBM entirely as a third level cache. Furthermore, we do not rely on explicit programming techniques to bind threads to specific NUMA nodes. Our way of programming guarantees fairness to comparison partners. We rely on the default hardware and compiler settings for other concepts, such as speculation or cache-coherence.

Chapter 3

Multi-core K-means

3.1 Introduction to a *Cache-conscious* Approach

To scale up data mining methods on current architectures, we need to re-engineer algorithms with the current hardware capabilities in mind entirely. As a showcase, we consider K-means clustering in a shared memory environment, such as current workstations or laptops. The main memory of current systems is usually large enough to occupy millions of data points. Therefore, we do not assume disk accesses. Parallel variants of K-means for distributed (shared-nothing) environments have been proposed [63, 135] and can be combined with our approach. We focus on exploiting MIMD and SIMD parallelism while optimally feeding each core with data along the memory hierarchy from registers to various cache levels. At first glance, this might seem an easy task since standards like Open-MP support MIMD parallelism, and SIMD parallelism is enabled by auto-vectorization performed by common compilers like GNU C++. However, if the algorithm's logical flow is not tailored to the opportunities and limitations of current architectures, we cannot expect significant performance gains.

In section 3.2, we revisit the K-means algorithm to fix the notation and outline the canonical implementation of the algorithm. We combine the canonical K-means algorithm with MIMD and SIMD parallelization techniques, including efficient use of available registers, outlined in section 3.3. We propose an elegant way to avoid branching in section 3.4, where we code current cluster memberships already in the data points. In our experiments, we compare the auto-vectorized technique and a K-means implementation based on Intel© BLAS [47] a quasi-standard for linear algebra operations in shared memory environments. In section 3.7 we give some concluding remarks.

3.2 K-means

To make this chapter self-contained, we introduce here the K-means algorithm for clustering n data points $x_0, ..., x_{n-1}$ from a d-dimensional vector space. Throughout this chapter, we will use the notation $x_i[\ell]$ for the ℓ-dimension $(0 \leq \ell < d)$ of data point x_i and $x_i[\ell, ..., \ell+3]$ for a sub-vector.The algorithm starts with random initialization of the cluster representatives $\mu_1, ..., \mu_k$ by k randomly selected points from the data set. Then, it repeats two steps until convergence: (1) assignment step: Each point $x_i \in \mathbb{R}^d$ from the data set is assigned to that cluster j which minimizes the Euclidean distance $||x_i - \mu_j||$, and (2) re-determination of the cluster centers: each cluster representative is computed as the center of mass of the associated points (centroid). Often, the collection of sufficient statistics (count and sum) for the re-determination step is already integrated with the assignment step. Thus, K-means is canonically implemented by four nested loops:

while not converged
  for $i := 0$ **to** $n{-}1$ ← consider x_i
    for $j := 0$ **to** $k{-}1$ ← compare it to μ_j
      for $\ell := 0$ **to** $d{-}1$ ← Euclidean dist. $||x_i - \mu_j||$

The ℓ-loop determines the Euclidean distance $||x_i - \mu_j||$. The j-loop determines the minimum among these distances, i.e. $\min_{0 \leq j < k} ||x_i - \mu_j||$ and additionally updates the sufficient statistics. A larger part of this chapter is devoted to changing this order of loop traversals to improve the transfer of information between main memory, cache, and registers.

3.3 Multi-core K-means

The main idea to parallelize our algorithm MKM (Multi-core K-means) is to enable MIMD parallelism at the level of data objects and SIMD parallelism at the level of dimensions. We distribute different contiguous subsets of the data objects to different threads running on different cores. We assign different dimensions of single objects and cluster representatives to different arithmetic/logic units within a core. For the implementation of MKM, we rely on Open-MP for MIMD and AVX1 for SIMD parallelism. For clarity, we base the following description on the common Ivy Bridge Processor Architecture characteristics with 16 registers of 256-bit size (4 double-precision floating-point vectors). However, the algorithm can easily be adapted to other micro-architectures.

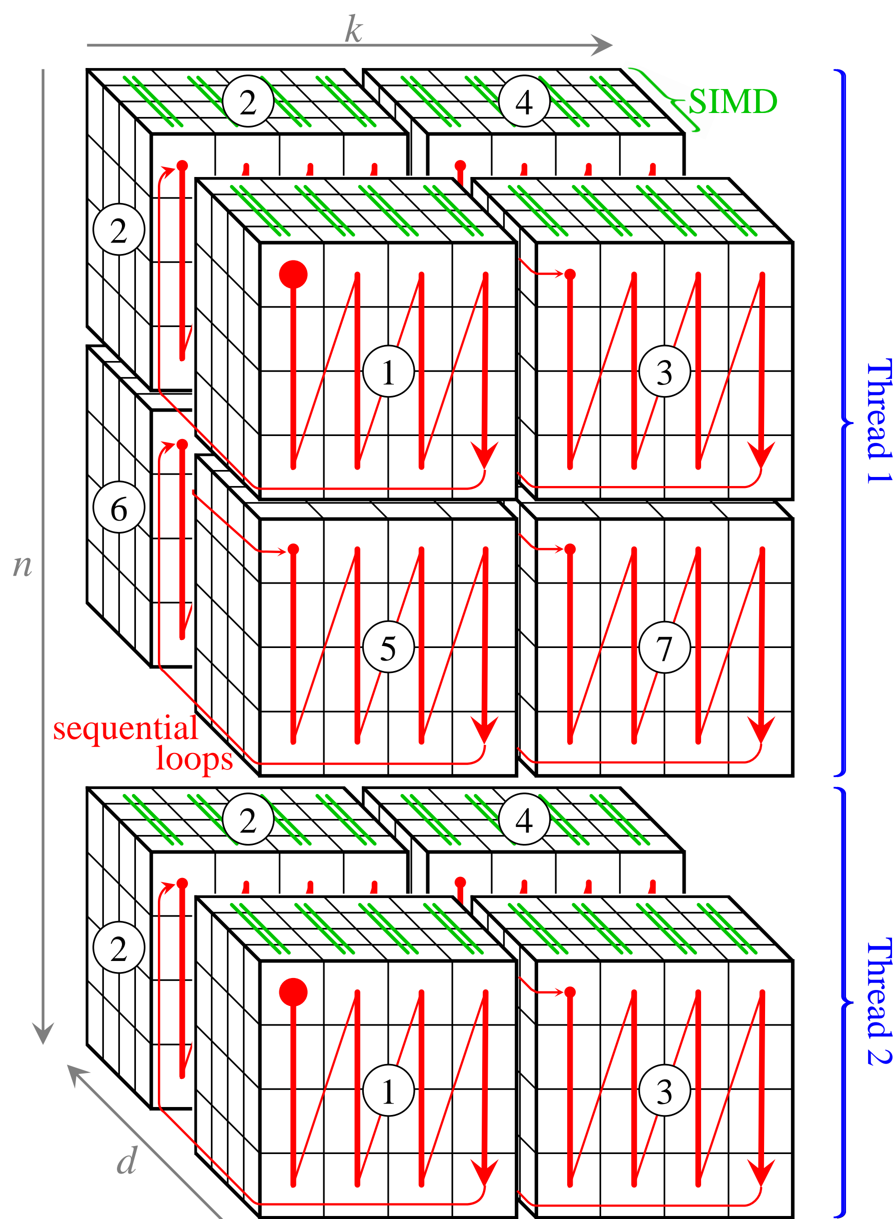


Figure 3.1: Loop traversal order over n, k, and d.

MIMD-parallelism

Our algorithm multi-core K-means assigns different parts of the data set to the different threads, i.e., contiguous blocks of objects are assigned and processed by the same core. If several c cores are available, we divide the data set into c, almost equally sized subsets. For reasons discussed in the next section, we take care that each subset contains a number

of objects divisible by four (and the last subset is enlarged by up to 3 vectors $[\infty, ..., \infty]$). If t is the thread number ($0 \leq t < c$), the start offset of the corresponding data subset is determined by $4 \cdot \lceil \frac{n \cdot t}{4 \cdot c} \rceil$. Additionally, each data vector is zero-padded to ensure that the data space's dimension is divisible by 4.

The easiest way of programming MIMD parallelism for multi-core processors is standards like Open-MP and CILK, where (among other possibilities) we have a special for-loop enabling parallel threads. In Open-MP, a usual for-loop in C-language syntax is prefixed by a compiler-hint ("#pragma omp parallel for"). Instead of executing a sequential loop, parallel threads are spawned (the number of which can be predefined by the user). Such loops have limitations, e.g., the loop iterator variable must not be modified in the loop body. Synchronization of memory access operations in the case of write dependencies is supported by critical sections (like "#pragma omp critical").

We use an Open-MP for-loop to parallelize the assignment step, which also collects the necessary statistics (sum and count of all assigned vectors) for K-means' next iteration. After this modified, MIMD-parallel assignment step, the threads are synchronized again, and a very efficient re-determination step computes from the collected statistics the new cluster representatives (centroids). MIMD parallelization of this re-determination (which is in $O(d \cdot k)$ time, constant in n) does not pay off, but SIMD parallelism can be used, as described later.

We avoid entirely write dependencies by ensuring that each thread writes only to private variables. During the run of the K-means algorithm, we have two types of write operations: (1) the cluster IDs as the intermediate result of the assignment step, and (2) the collection of the statistics (sum and count of all assigned vectors) for the next K-means iteration. Since we assign blocks of contiguous objects to each thread, we implicitly assign a block of contiguous cluster-IDs to each thread, and therefore, we have no write conflicts in (1). The collection of statistics (2) is done in private variables for each thread, which causes a small memory overhead of $O(d \cdot k \cdot c)$ and the same time complexity for the (non-parallel) consolidation of the private variables into global variables.

Efficient Use of SIMD Registers.

AVX1 offers us a relatively high number of 16 registers (called YMM_0 to YMM_{15}) for SIMD operations, each of which can store up to 4 double-precision floating-point numbers (64 values in total). For high performance, it is essential to minimize data transfer from

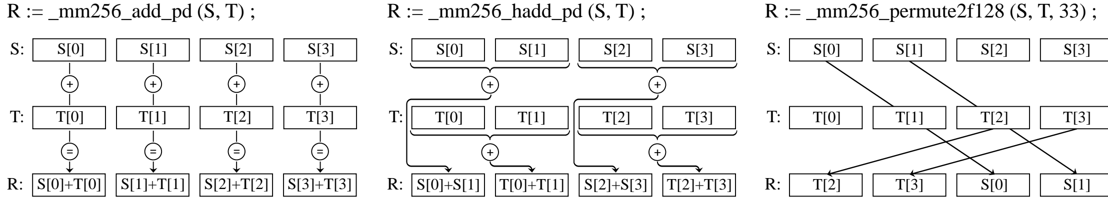


Figure 3.2: Vertical and Horizontal Addition, Permutation of Vectors.

and to these registers. As we want to avoid any restrictions on $n, k,$, and d, we cannot safely assume that whole points or centroids fit into the registers. Therefore, our strategy uses many of the registers (YMM_0 to YMM_7) to store the intermediate results of 16 distance calculations between four data points (now called the *current points*) and four *current centroids*, respectively. Although it might seem sufficient to use four registers to store these 16 values, we will discuss later a performance issue why we use two variables for each distance, marked by “a” and “b”. Moreover, we use five registers, each of which stores four subsequent dimensions of the current data points and the current centroids. We need two further registers for intermediate results and one register (YMM_{15}) for the final minimal distances used in the assignment operation of the four current points. The following table gives us an overview, where $x_0, ..., x_3$ are the four current points and $\mu_0, ..., \mu_3$ are the four current centroids (actually standing for $x_i, ..., x_{i+3}, \mu_j, ..., \mu_{j+3}$), and $\ell, ..., \ell + 3$ are the four current dimensions:

$$
\begin{aligned}
\text{YMM}_0 &= \big[||x_0-\mu_0||_\text{a}^2, ||x_1-\mu_0||_\text{a}^2, ||x_0-\mu_0||_\text{b}^2, ||x_1-\mu_0||_\text{b}^2\big]\\
\text{YMM}_1 &= \big[||x_2-\mu_0||_\text{a}^2, ||x_3-\mu_0||_\text{a}^2, ||x_2-\mu_0||_\text{b}^2, ||x_3-\mu_0||_\text{b}^2\big]\\
\text{YMM}_2 &= \big[||x_0-\mu_1||_\text{a}^2, ||x_1-\mu_1||_\text{a}^2, ||x_0-\mu_1||_\text{b}^2, ||x_1-\mu_1||_\text{b}^2\big]\\
&\vdots\\
\text{YMM}_7 &= \big[||x_2-\mu_3||_\text{a}^2, ||x_3-\mu_3||_\text{a}^2, ||x_2-\mu_3||_\text{b}^2, ||x_3-\mu_3||_\text{b}^2\big]\\
\text{YMM}_8, ..., \text{YMM}_{11} &= x_0, ..., x_3 \text{ (current 4 dimensions)}\\
\text{YMM}_{12} &= \mu_0, \text{ later: } \mu_1, \mu_2, \mu_3 \text{ (current 4 dimensions)}\\
\text{YMM}_{13}, \text{YMM}_{14} &: \text{ reserved for intermediate results.}\\
\text{YMM}_{15} &= \big[\min_j ||x_0 - \mu_j||, ..., \min_j ||x_3 - \mu_j||\big]
\end{aligned}
$$

Using this storage scheme, we can efficiently make use of all 16 available SIMD registers. For this purpose, we have now to change our original loop scheme of K-means involving four loops into a new one involving eight loops, of which one (marked in blue) is an Open-MP loop operating MIMD parallel threads rather than sequential iterations. The

innermost (green) loop performs SIMD-parallel operations (multiply and add), and two other innermost loops (iterating over I and J) are not implemented as loops. However, they are instead explicitly programmed in an unrolled way in the C-language source code and in the following pseudo-code. The full hierarchy of 8 loops exactly represents how data objects and cluster representatives are accessed and processed. This order is also visually depicted in Figure 3.1, where the colors are consistent with the text colors in our eight loops: the green connected parts are simultaneously processed using SIMD parallelism, and the two simultaneous threads of MIMD parallelism are braced together in blue:

```
while not converged
  parallel threads for t := 0 to c − 1                  ← MIMD
    for i := 4⌈nt/4c⌉ to 4⌈n(t+1)/4c⌉ − 1 step 4
      for j := 0 to k − 1 step 4
        for ℓ := 0 to d − 1 step 4
          for J := j to j + 3                          ← unrolled
            for I := i to i + 3                        ← unrolled
              for L := ℓ to ℓ + 3                      ← SIMD
```

In Figure 3.1, each of the cubic blocks represents the processing of four current points and four current centroids in four current dimensions (three innermost loops). Note that the four current dimensions are processed by SIMD parallelism. The traversal pattern within each cubic block is generated by the two unrolled loops (J, I). The traversal order across blocks is established by the three loops $i, j,$, and ℓ (additionally marked by numbers).

The assembler instructions to process information in SIMD-registers have been mapped to higher programming languages like C. "Intrinsic operations" like _mm256_add_pd and _mm256_mul_pd often use two SIMD registers as operands and store the result in a third register, as demonstrated for the "add"-operation on the left side of Figure 3.2 where R, S, and T are any of the registers $\text{YMM}_{0,\ldots,15}$, and $R[0], \ldots, R[3]$ denote the 4 dimensions.

We can jointly load four subsequent values of x_i and μ_j into registers, add, and multiply them for the Euclidean distance. However, the horizontal sum (the sum of several components *inside* a register) of these four intermediate results, which is finally needed for the Euclidean distance, is a bit more tricky. AVX offers an operation called "hadd"

(horizontal add), which can add only the first two and the last two vector components, respectively, for two source registers (cf. Figure 3.2, center).

The horizontal sum between the second and third component (e.g.) requires operations like *blend* and *permute* (Fig. 3.2 right side), causes more effort and is thus not done in the innermost loop (see later). It is better to defer this operation and to store instead of the two partial sums in independent components of the registers, although we are thus wasting 4 of the valuable registers. The three innermost of our eight loops are implemented in an unrolled way as follows (the indices in red color represent the unrolled loops (I, J, L) and help to figure out modifications in the three repeats of the sequence $(*)$):

for $\ell := 0$ **to** $d-1$ **step** 4

$\mathrm{YMM}_8 := x_{i+0}[\ell, ..., \ell+3];$

$\vdots$ likewise $\mathrm{YMM}_{9,...,11} := x_{i+1,...,3}[\ell, ..., \ell+3];$

$$\left.\begin{array}{l}
\mathrm{YMM}_{12} := \mu_{j+0}[\ell, ..., \ell+3];\\
\mathrm{YMM}_{13} := _\mathrm{mm256_mul_pd}(\mathrm{YMM}_8, \mathrm{YMM}_{12});\\
\mathrm{YMM}_{14} := _\mathrm{mm256_mul_pd}(\mathrm{YMM}_9, \mathrm{YMM}_{12});\\
\mathrm{YMM}_{13} := _\mathrm{mm256_hadd_pd}(\mathrm{YMM}_{13}, \mathrm{YMM}_{14});\\
\mathrm{YMM}_{0} := _\mathrm{mm256_add_pd}(\mathrm{YMM}_{0}, \mathrm{YMM}_{13});\\
\mathrm{YMM}_{13} := _\mathrm{mm256_mul_pd}(\mathrm{YMM}_{10}, \mathrm{YMM}_{12});\\
\mathrm{YMM}_{14} := _\mathrm{mm256_mul_pd}(\mathrm{YMM}_{11}, \mathrm{YMM}_{12});\\
\mathrm{YMM}_{13} := _\mathrm{mm256_hadd_pd}(\mathrm{YMM}_{13}, \mathrm{YMM}_{14});\\
\mathrm{YMM}_{1} := _\mathrm{mm256_add_pd}(\mathrm{YMM}_{1}, \mathrm{YMM}_{13});
\end{array}\right\} (*)$$

$\vdots$ $(*)$ is $3\times$ repeated with following modifications:

$\vdots$ $\mathrm{YMM}_{12} := \mu_{j+1}[\ell, ..., \ell+3];$ add to $\mathrm{YMM}_{2 \text{ and } 3};$

$\vdots$ $\mathrm{YMM}_{12} := \mu_{j+2}[\ell, ..., \ell+3];$ add to $\mathrm{YMM}_{4 \text{ and } 5};$

$\vdots$ $\mathrm{YMM}_{12} := \mu_{j+3}[\ell, ..., \ell+3];$ add to $\mathrm{YMM}_{6 \text{ and } 7};$

Altogether, we have a number $4 \cdot \lceil \frac{d}{4} \rceil$ of load operations for centroids (from L1-cache), $4 \cdot \lceil \frac{d}{4} \rceil$ of load operations for data points (from main memory), $16 \cdot \lceil \frac{d}{4} \rceil$ AVX-multiplications, $8 \cdot \lceil \frac{d}{4} \rceil$ additions, and $8 \cdot \lceil \frac{d}{4} \rceil$ “hadd”-operations. After finishing this ℓ-loop our registers $\mathrm{YMM}_{0,...,7}$ contain the 16 scalar products of the four current points $x_i, ...x_{i+3}$ with the four current centroids $\mu_j, ..., \mu_{j+3}$. The horizontal sums must then be completed, and the scalar products must be changed into Euclidean distances using the following operations:

YMM_{13} := _mm256_blend_pd(YMM_0, YMM_1, 12);
YMM_{14} := _mm256_permute2f128(YMM_0, YMM_1, 33);
YMM_{14} := _mm256_add_pd(YMM_{13}, YMM_{14});
YMM_{13} := _mm256_broadcast_sd(scalar[$j+0$]);
YMM_{13} := _mm256_sub_pd(YMM_{13}, YMM_{14});

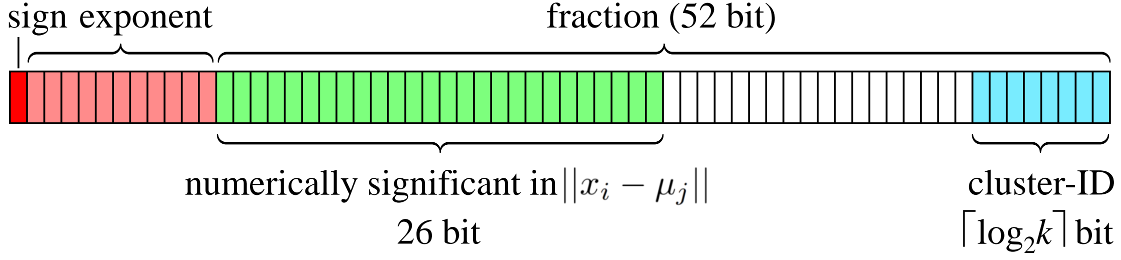
where the constants 12 and 33 are masks to control the blend, and permute operations and scalar is a pre-computed array containing the self scalar products $\langle \mu_j, \mu_j \rangle$ for all $0 \leq j < k$. The broadcast operation sets all components of the register to the same value. After the subtraction YMM_{13} contains a vector of four values which are monotonic with the Euclidean distances:

$$[||x_i - \mu_j||, ||x_{i+1} - \mu_j||, ||x_{i+2} - \mu_j||, ||x_{i+3} - \mu_j||]$$

Furthermore, YMM_{13} can be compared to the vector YMM_{15}, which stores in each component, the minimal previously found distance to a centroid (i.e., among all centroids $\mu_{j'}$ with $j' < j$). We use the component-wise minimum operation _mm256_min_pd, which simultaneously compares the four components of YMM_{13} to those of YMM_{15}.

3.4 Cluster ID Coding

However, our goal here is to keep track of the minimum distance for each point of x_i but also for the cluster-ID j, which caused the minimum distance. More formally, the main result of the assignment step of the K-means algorithm is for each point x_i the cluster-ID, which is $\min_j ||x_i - \mu_j||$. Our idea to achieve this with clever use of the AVX SIMD operations and without any expensive branching operations (which cause a break in the processor's operation pipeline if the branch prediction fails) to code the cluster-ID directly in the distance value. According to the IEEE-754 specification, double-precision floating-point values are represented using a 52-bit fraction (sometimes also called the mantissa), an 11-bit exponent, and a sign (1 bit), cf. Figure 3.3. For cluster-ID coding, we use a number $\lceil \log_2(k) \rceil$ of the least significant bits of the fraction. This coding does not change the distance value significantly anyway (if $k = 8$ and the distance is 1.0, our ID coding changes the distance value to 1.000000000000001). Nevertheless, a numerical



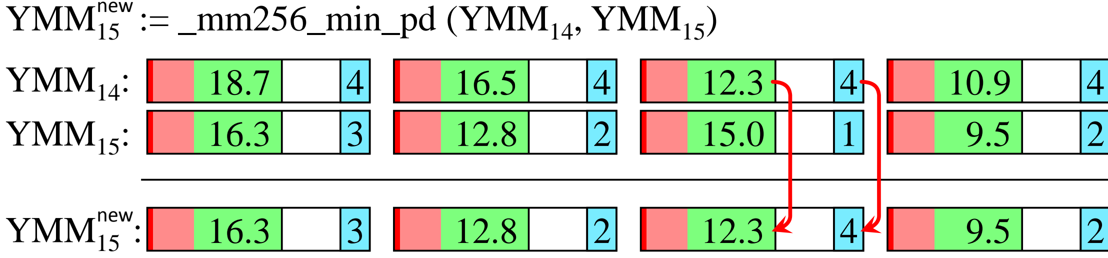


Figure 3.3: Cluster ID coding in IEEE-754 format.

analysis of the Euclidean distance reveals that only roughly half of the fraction bits are numerically significant (in our example, the distance is only known to be between 1.0 and 1.000000015). Therefore, even for unrealistically large k in the order of a million ($\approx 2^{20}$) our coding affects only that part of the fraction, which is numerically insignificant and thus filled with random bits. We use the AVX operations for bitwise logic operations ("and", "and not", "or") for our cluster-ID coding (including the final minimum-operation):

YMM_{14} := _mm256_broadcast_sd($2^{\lceil \log_2(k) \rceil} - 1$);
YMM_{13} := _mm256_andnot_pd(YMM_{14}, YMM_{13});
YMM_{14} := _mm256_broadcast_sd($j + 0$);
YMM_{14} := _mm256_or_pd(YMM_{14}, YMM_{13});
YMM_{15} := _mm256_min_pd(YMM_{15}, YMM_{14});

The operation _mm256_broadcast_sd sets all four components to the same value, a globally constant mask with those bits set that have to take over the cluster-ID.

The impact of the final operation _mm256_min_pd is visualized in the lower part of Figure 3.3: Both YMM_{14} and YMM_{15} contain four distance values with backpacked cluster IDs each (the color scheme corresponds to the upper part of Figure 3.3 but we

now use the decimal representation of distances and cluster IDs). Applying the min-operation leads to a modification of the third component of YMM_{15}, the only component in which YMM_{14} is less than YMM_{15}. But the minimum operation does not only copy the distance (12.3) from YMM_{14} to YMM_{15} but also the corresponding cluster-ID (4). The consolidation step for the Euclidean distance and the coding step for the cluster-ID is also repeated for the comparison of the four current data points to μ_{j+1}, (stored in the registers YMM_2 and YMM_3), μ_{j+2} (YMM_4, YMM_5), and μ_{j+3} (YMM_6, YMM_7), respectively. After finishing the j-loop, YMM_{15} contains the final minimum distances and the corresponding coded cluster IDs, which can be extracted again using the mask operation:

$$\text{YMM}_{14} := \text{_mm256_broadcast_sd}(2^{\lceil \log_2(k) \rceil} - 1);$$
$$\text{YMM}_{15} := \text{_mm256_and_pd}(\text{YMM}_{14}, \text{YMM}_{15});$$
$$x_{i,\ldots,i+3}.\text{CID} := \text{YMM}_{15};$$

These 4 cluster IDs are then immediately used to update the statistics (count and sum) in private variables for each thread, again using the AVX operation _mm256_add_pd. Note that this step causes no branching either. Therefore, apart from loops, our algorithm is completely free from branching statements like *if*, *switch*, etc.

Equivalence to Standard K-means

The assignment step is guarantees to assign each point x_i to $\min_j ||x_i - \mu_j||$, identically for MKM and standard K-means. Each thread collects the sufficient statistics locally, but they are consolidated before using them in the next iteration. Therefore, in every iteration, we obtain the same centroids as standard K-means. Our algorithm MKM is equivalent to standard K-means. If we assume a common initialization for the centroids, both algorithms converge to the same result requiring the same number of iterations.

3.5 Experiments

Microarchitecture. All experiments were performed on a quad-core CPU E5-2609 (Sandy Bridge micro-architecture) featuring AVX1. This CPU has a cache size of 4×32 KB (L1 instruction cache), 4×32 KB (L1 data cache), 4×256 KB (L2 cache), and 10 MB (shared L3 cache). The latency for load operations from L1 cache to registers is 4 clock

cycles, with a load bandwidth of 32 bytes per cycle. Reading from L2 cache has a latency of 11 clock cycles and a bandwidth of 32 bytes per cycle. Our workstation has two such CPUs on shared memory, so 8 cores are maximally available. All algorithms have been implemented in C++ and compiled with the GNU g++ compiler version 4.7.1. This compiler uses the current SSE version as default. The usage of AVX must be specified as a compiler flag.

Data. The synthetic experiments have been performed on randomly generated data with $d = 20$ and $k = 40$ whenever not otherwise specified. The number of iterations in K-means depends on the initialization and the structure of the optimization surface. It can vary greatly even for data originating from a typical data distribution but sampled with different numbers of objects. Although the number of iterations is always guaranteed to be identical when comparing all versions of multi-core K-means with standard K-means within a given configuration (i.e. number of objects n, number of clusters k, number of dimensions d, etc.) we want also to facilitate comparison across configurations and we therefore report the time required for 5 iterations whenever not otherwise specified. Besides synthetic data, we used two real data sets from the UCI Machine Learning Repository [78]. In our experiments on real data, we run MKM until convergence to get an impression of the practice's runtime behavior. The Forest Covertype data consists of 581,012 instances characterized by 54 attributes. The instances are labeled into 7 classes representing different forest cover types. The Household data consists of 2,049,280 instances described by 7 numerical attributes. We removed instances with missing values from the original data. For Covertype, we set k to the number of classes and clustered the Household data with $k = 8$.

Comparison to Standard K-means.

To see how much performance gain we can achieve by exploiting MIMD and SIMD parallelism, we compare our algorithm MKM to the classical K-means algorithm implemented in C++. Classical K-means has been implemented in four variants: as a single-threaded as well as a multi-threaded version, and with compiler options on and off, respectively, allowing the compiler to use AVX-auto-vectorization. We refer to these variants as "No Vectorization" and "Auto-vectorization", and additionally specify the number of threads for MIMD parallelism, e.g., in Figure 3.4 the numbers 1 (for "No MIMD parallelism") and 8 (for the maximum useful MIMD parallelism on eight cores of two processors). The MIMD parallel versions of standard-K-means distribute, like our own algorithm MKM, contiguous parts of the data set to the different cores. In Figure 3.4, we can recognize that

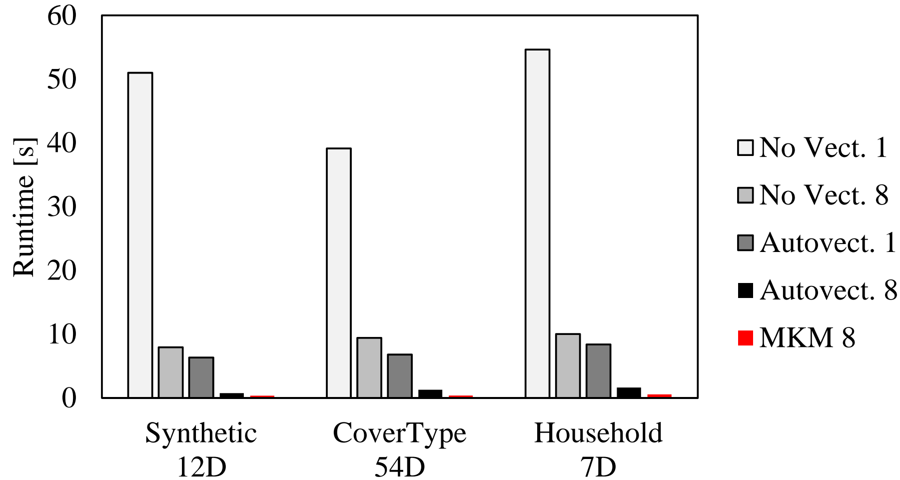


Figure 3.4: Comparison to Standard K-means.

standard K-means with no parallelism needs between 39 and 55 seconds to cluster our three data sets. In this experiment, we report the total runtime until convergence. Using MIMD parallelism alone reduces this time by a factor between 4.1 and 6.4, and likewise does SIMD-parallelism alone. The combination of SIMD and MIMD parallelism reduces the runtime to 0.7 to 1.6 seconds. However, this is still clearly outperformed by our new algorithm MKM, which needs only 0.3 to 0.5 seconds. We observe this behavior on synthetic 12-D data and similarly also on the Covertype and the Household data, which demonstrate the benefits of MKM in practice. Our algorithm enables K-means clustering large data sets with millions of points in milliseconds on a standard workstation.

Speed-up for Varying Number of Cores.

As Auto-vectorized K-means on multiple cores were the strongest comparison partner, in our following experiments, we compare our algorithm MKM to standard-K-means with Auto-vectorization using the same amount of parallel threads as MKM (usually 8). To systematically vary n, d, and k, we focus these experiments on randomly generated synthetic data and fix the number of K-means-iterations to 5. First, we examine the behavior of the algorithms with a varying number of threads or cores (speedup). Figure 3.5 displays the run-time of 5 iterations of the synthetic dataset ($d = 20, k = 40, n = 64$ Million = 10 GByte). We also plotted the expected run-time in dashed lines when assuming an ideal, linear speedup with the number of cores. We can recognize that both MKM and standard-K-means are close to the linear speedup. However, MKM being faster than standard-K-means by a factor between 5.3 and 6.9. Generating more than eight threads

did not lead to further improvements since only eight cores were available.

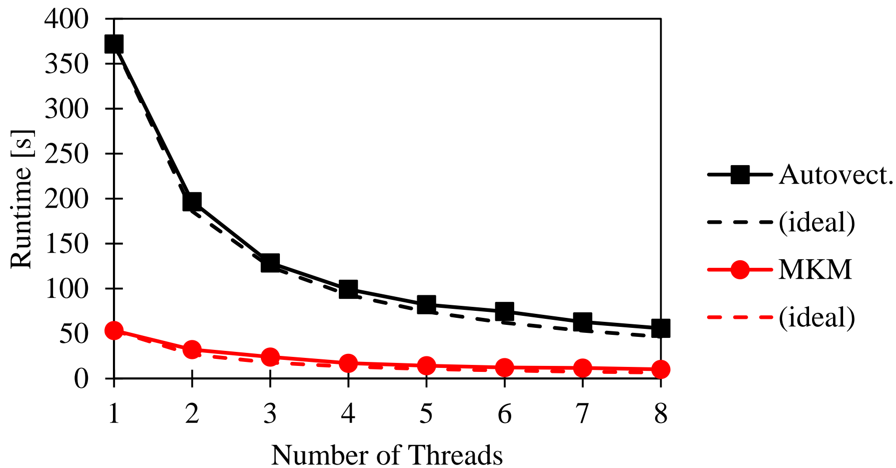


Figure 3.5: Speedup with Varying Number of Threads.

Evaluation of Cluster-ID Coding.

In Figure 3.6 we evaluate the impact of cluster-ID coding as described in Section 3.4 on the overall performance on the same data as used in speed-up experiment (cf. Figure 3.5). We compare our algorithm MKM, which applies cluster-ID coding (backpacked on the numeric representation of the distance) to the same algorithm without ID coding. The comparison algorithm stores the minimum distance and the corresponding cluster-ID in separate variables and performs for each distance comparison an IF-THEN-ENDIF statement (involving a conditional jump in the compiled machine code) to update both. MKM is by a factor 1.5-1.9 faster with cluster-ID coding, which is remarkable as intuitively, a comparison seems much cheaper than the distance calculation. Probably, the conditional jump is difficult to predict and, therefore, expensive.

Scalability with Varying n, d, and k.

In our last set of experiments, we varied n, d, and k with the standard-setting $n = 32$ Million, $d = 20$, $k = 40$. We can recognize that both algorithms have a close-to-linear runtime in all these parameters. MKM constantly outperforms the auto-vectorized K-means by a factor of 5.4 (averaged over all experiments).

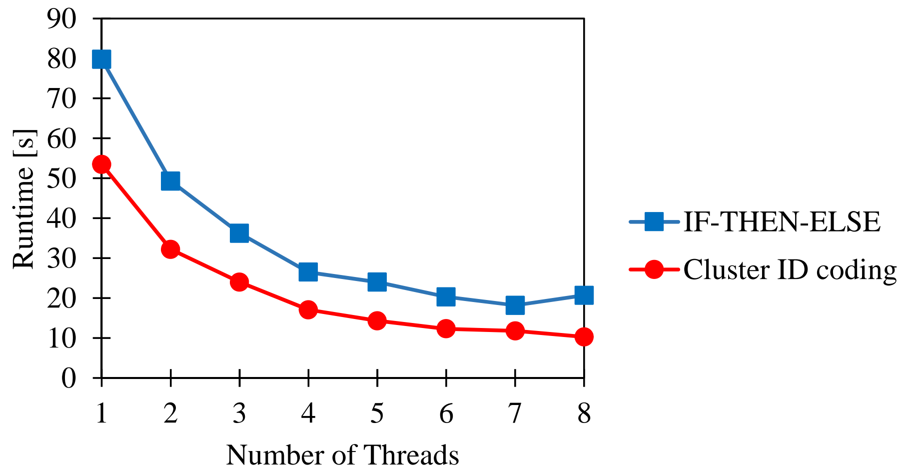


Figure 3.6: Effect of Cluster-ID Coding.

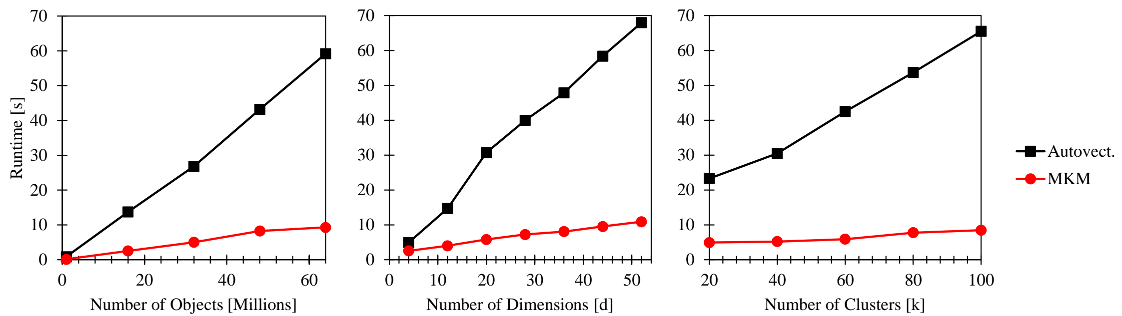


Figure 3.7: Scalability of the Different Strategies with n, d, and k.

3.6 Related Work and Discussion

Due to the success of K-means there has been a lot of research effort on scaling the algorithm to massive data. We classify existing approaches into the following categories:

- Algorithms for special environments, e.g., computing clusters [135],
- indexing approaches, mostly based variants of the kd-tree, e.g.,
- different algorithmic approaches allowing a fast approximation of the K-means result, e.g. [116].

K-means for particular environments. The first category of techniques is most related to our approach. Most of the existing approaches focus on distributed environ-

ments and are often based on MapReduce, e.g., [1, 63, 135]. Usually, the mappers process different parts of the data, and the reducers perform the update step. The algorithms differ in the exact implementation of both steps, e.g., the technique [1] involves a combiner process which locally aggregates the sufficient statistics received from different mappers. Besides considering the general setting of distributed environments, K-means has been investigated in the context of specific hardware, e.g., K-means for Graphic Processing Units [77] or supercomputers [31]. Other approaches even design hardware architectures [37] or combined hardware- and software architectures [5] for the sole purpose of efficient K-means clustering. Orthogonal to these approaches for very specific architectures, we consider how to scale up K-means clustering on a current workstation. Standard workstations are equipped with a multi-core CPU supporting inter-core MIMD parallelism and inner-core SIMD parallelism. Moreover, we can assume that the data fits into the main memory. For example, 8 GB of main memory is sufficient to occupy 10 million 100-dimensional feature vectors. Hadian and Shahrivari [58] also study the setting of a single multi-core machine but with the assumption that the data does not fit into main memory. Similarly to our algorithm MKM, their algorithm splits the data into chunks that are distributed to the cores. However, their work differs from ours in three central aspects: Firstly, Hadian and Shahrivari do not consider SIMD parallelism. Secondly, their technique minimizes random disk accesses regarding memory latency, i.e., page faults, while we aim to avoid cache misses. Thirdly, in contrast to MKM, their algorithm is not equivalent to standard K-means in the sense that it converges to the same result as K-means starting from a common initialization. Inspired by the initialization strategy K-means++, the algorithm [58] considers the centroids of the chunks as an approximation of the corresponding part of the data. The results are then merged by a master thread performing a run of K-means on the chunk centroids. In contrast to MKM, this approach also belongs to categories (2) and (3) mentioned above.

Indexing. As a large part of the runtime in K-means is spent on distance calculations, index structures have been proposed to accelerate K-means. Most approaches are based on the kd-tree, which is a simple index structure suitable for main memory, which is applied, e.g., in the techniques [58, 103]. Each node of the kd-tree is represented by a bounding box specifying the minimal axis-parallel hyper-rectangle containing all associated points. Based on the bounding box and the current centroid locations, some of the centroids can be excluded for all node data points. Elkan proposed to bound the distances between points and centroids with the triangle inequality [49]. As a space partitioning index structure the kd-tree tends to degrade in high-dimensional spaces. At the

same time, the pruning based on the triangle inequality has been experimentally demonstrated to work up to 1000 dimensions. We apply SIMD parallelism to speed up distance calculations, a strategy that does not depend on the distribution or the dimensionality of the data. Moreover, SIMD parallelism can be integrated into indexing strategies in future work.

Approximations of K-means. Only marginally related to MKM are algorithms that aim at a fast approximation of the K-means result. One example is the already discussed algorithm by Hadian and Shahrivari [58], which is based on K-means++ followed by re-clustering. Another example that deviates even more from classical K-means is the approach of Shindler et al. [116]. Assuming a streaming environment, the authors propose a technique based on the online facility location algorithm combined with an approximate nearest neighbor search. Contrary to these approaches, our method provides quality bounds on the K-means objective function.

Vectorization. Large parts of the performance gain of MKM over standard K-means is due to SIMD parallelism. In current practice, auto-vectorization by the compiler is the most common access to the SIMD instruction sets. However, the achievable speed-up strongly depends on the application developer's experience as it requires a specific loop- and data structures. Only for certain types of rather simple loops, the compiler can decide if they support safe vectorization. Therefore, algorithms require comprehensive re-engineering. Recently, SIMD accelerated algorithms for specific computationally demanding tasks have been proposed, e.g., index compression algorithms for similarity search [139], integer factorization algorithms for cryptography [112] or particle simulation for astrophysics [96]. Besides these particular and highly optimized algorithms, only a few more general studies have been performed. Ladra et al. demonstrated that basic string algorithms, which are building blocks for many applications like ranking, selection, and suffix trees, effectively support direct usage of the intrinsics provided by SSE4.2 [74]. We position our work in between the highly optimized complex algorithms for specific tasks and very basic algorithms by contributing building blocks accelerating common data mining algorithms on vector data, most importantly the efficient implementation of the *argmin* function.

3.7 Conclusion

In this chapter, we have introduced the algorithm MKM combining inter-core MIMD parallelism with intra-core SIMD parallelism of current microarchitectures for efficient K-means clustering. Accessing the computing power of current microarchitectures requires major re-engineering of the classical K-means algorithm. We optimized the logical flow of MKM to enable MIMD and SIMD parallelism and to minimize the memory latency. Moreover, by an innovative coding strategy, we continuously keep track of an object's cluster assignment, enabling us to perform K-means clustering without any branching or case distinction. The massive performance potential of our algorithm MKM combining SIMD and MIMD parallelism is demonstrated in the experiments with an acceleration of 95–142 over non-parallel standard K-means and 4–5 over auto-vectorized multi-threaded standard-K-means and a SIMD and MIMD parallel K-means variant based on BLAS.

In the following, we focus on space-filling curves and how they could be used to exploit the memory hierarchy in today's shared memory systems in the chapter. Especially we will look at the Hilbert curve in chapter 5 and at the Morton-order curve in chapter 8. However, before we go into the details, we introduce some preliminaries to space-filling curves in general in chapter 4.

Chapter 4

Construction of Space-Filling Curves

This chapter elaborates on previous methods to construct space-filling curves and their connections to the work presented in this thesis. This is far from being a complete reference for all the research done in this field. Moreover, we summarize the primary milestones and guide the reader to understand the context of our work. Furthermore, we give an introductory example to L-systems (or Lindenmayer systems), a grammar that is used in the construction of our Hilbert curve. For a more detailed explanation of space-filling curves' mathematical and analytical cornerstones, we refer the interested reader to Sagan [108]. Beyond that, Bader [10] connects space-filling curves thematically with computer science in general and gives excellent examples for space-filling applications curves in the context of scientific computing.

4.1 Introduction and Historical Context

The term "curve" goes back to 1887, where Camille Jordan introduced the rigorous definition of a curve, which has been adopted very often in various domains:

> A curve (with endpoints) is a continuous function whose domain is the unit interval [0,1].

In the most general form, the application is a topological space, and the interval will lie in a Euclidean space, such as the 2-dimensional plane (a planar curve) or the 3-dimensional space (space curve). The common understanding was that such a curve is piecewise differentiable and cannot fill up an entire unit square. In 1879 Eugen Netto

proved that a curve could not be bijective and continuous at the same time. However, in 1980 Guiseppe Peano [101] and David Hilbert [64] presented curves, that are continuous and surjective (but not injective). Guiseppe Peano introduced a continuous curve that passes every point of the unit square [101], which is now known as the Peano curve (see Figure 1.1b). As this curve visit every point of the unit square, it is called *space-filling*. The idea that a one-dimensional curve may completely cover an area or a volume was, at that time, completely novel and counter-intuitive. More examples followed after Peano's publication such as the Koch curve [72], the Cantor Set [32, 117] or the Hilbert curve [64]. As a result, space-filling curves have been studied by many highly influential mathematicians, such as Peano, Hilbert, Lebesgue, Sierpinski to name only a few. The book of Hans Sagan [108] summarizes these curves very well. It provides an excellent introduction and overview of the mathematical aspects of space-filling curves and their history. Space-filling curves such as the Hilbert curve are often limited to $n \times n$ matrices, where n is any power of 2 in cases of the Hilbert curve or the Morton-order curve or in case of the Peano curve n is limited to the power of 3. We address this limitation in our thesis, but throughout this chapter, we will ignore this limitation and revisit the construction of space-filling curves to understand better how space-filling curves are constructed and how to address this issue.

We introduce a sequential order on a d-dimensional array of elements (or cells). Since we restrict ourselves to the 2-dimensional case, this is nothing more than a 2-dimensional mapping between a range of array indices $\{1, ..., n\}^2$ to sequential indices $\{1, ..., n^2\}$. In computer science, two- or multidimensional data structures are ubiquitous. They are present in vectors, matrices, tensors in linear algebra, all kinds of rasterized images, coordinates in general and statistical data in various fields in terms of graphs, baskets, or time series analysis. These multidimensional data structures need a form of sequentialization. This could be a straight forward traversal of the data, storing or retrieving data from main memory, or introducing a sequential ordering through sorting. This sequential ordering can be expressed as two nested loops

for $i := 0$ **to** $n - 1$ **do**

  for $j := 0$ **to** $n - 1$ **do**

or similar constructs as a traversal of 2- or multidimensional arrays. However, what are the necessary properties to traverse such a data structure? A necessary property is to

generate a 1 to 1 mapping between the 2 dimensional data structure and the sequential mapping, such that all data items are visited exactly once. In a mathematical sense, this would be a *bijective* mapping. Moreover, skipping some indices is not allowed, such that we would need a *continuous* mapping as well. According to the definition of Jordan, the term "curve" is quite appropriate. In mathematical analysis, a space-filling curve is a curve whose range contains the entire 2-dimensional unit square (or more generally an n-dimensional unit hypercube).

In the following, we illustrate common approaches to construct space-filling curves. For this purpose, we will use a prominent example of a space-filling curve, namely the Hilbert curve. Hilbert was the first to recognize a general geometrical generating procedure that allowed the construction of an entire class of space-filling curves. The construction of the Hilbert curve is based on a recursive procedure:

1. The squared matrix is divided into four sub-matrices, each with a side length of half of the parent matrix.
2. For each of the sub-matrices, we need to find a rotated or reflected version of the original curve.
3. The rotation and reflection of the sub-matrix need to preserve the continuity of the matrix so that the partial curves can be connected to each other parts.

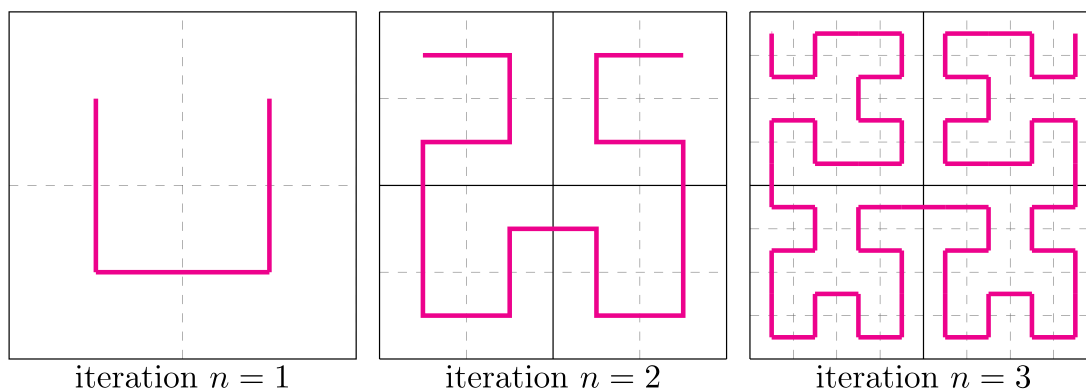


Figure 4.1: Iterations of the Hilbert curve.

These reflections and rotations can be found again if one considers the iterations of the Hilbert curve. The iterations describe a recursive building principle. The initial

pattern of the first iteration ($n = 1$) is called a "Leitmotiv". In the following iteration ($n = 2$) the next pattern is formed based on the previous iteration, such that all matrices of the current iteration share a common consecutive edge. Figure 4.1 outlines the first three iterations. One important detail here is that a sub-matrix shares a common edge with the exit point on this sub-matrix at each entry point of the curve. This detail combined with the concept of iterations is used in the following to construct the Hilbert curve based on approximating polygons.

4.2 Hilbert Function On the Example of Approximating Polygones

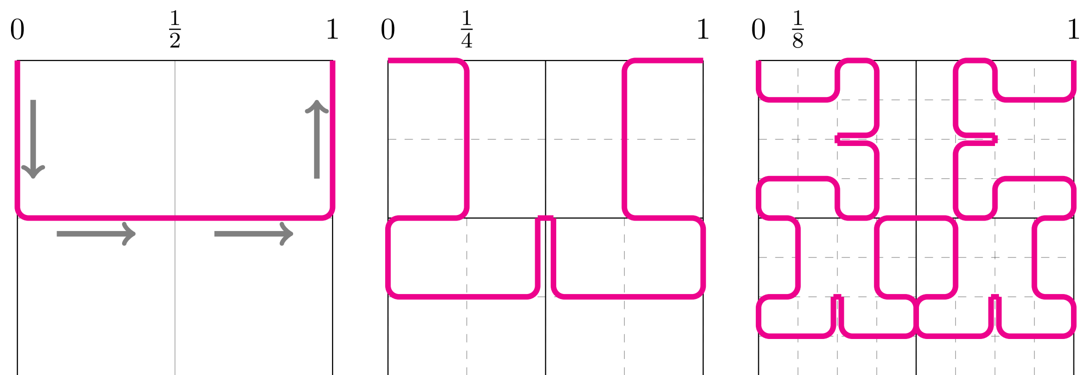


Figure 4.2: First three approximations of the Hilbert curve using polygons.

The idea behind the construction with approximating polygons is to assume the matrix has a side length of 1. The matrix contains 2^{2n} cells, each with a side length of $\frac{1}{2}^n$. Instead of approximating the cells that are visited directly as depicted in Figure 4.1, one could also approximate the polygonal line that lies in the sub-matrix. The first three approximating polygons are presented in Figure 4.2. If we take a detailed look at the sub-matrices (solid lines), and in particular if one refines the iterations accordingly, we see that this also holds for the sub-matrices: the Hilbert curve will enter each sub-square in one particular corner, and will exit the sub-matrix in one of the corners that share a common edge with the entry corner (arrows at the first iteration). If we connect these entry and exit corners with a polygon, we obtain the so-called approximating polygon of a Hilbert curve. Hilbert explained his geometrical generation procedure on basis of an interval $\mathcal{I} = [0, 1]$ which are the approximating polygons. The sequentialisation approach is then formulated as a function $h = \mathcal{H}(t)$, where f_h is a surjective mapping function. The function $\mathcal{H}$ maps $\mathcal{I} \rightarrow \mathbb{E}^2$. $\mathbb{E}^2$ is the 2-dimensional Euclidean space $\mathbb{R}^2$ including the

Euclidean norm which defines the metric. The parameter $t \in \mathcal{I}$ indicates several time points in the space. This means $\mathcal{H}(0)$ is called the beginning of the Hilbert curve and $\mathcal{H}(1)$ is called its endpoint.

The most important aspect related to our work is mainly how functions are used in the mathematical and analytical context so far. Here, the function $\mathcal{H}$ is used to obtain a sequential order from the Hilbert order. In the use case of linear algebra, there is an incremental variable e.g., the Hilbert value, which indicates the current position in our Hilbert curve. The function requires to deliver the i and j variables to access the array. This is the reason for using the inverse function $\mathcal{H}^{-1}$ for describing the Hilbert curve in section 5.4 or the Morton order curve in section 8.2. In the following, we will look at a decoding algorithm $\mathcal{H}^{-1}$ of the Hilbert curve, which returns the coordinates x and y, respectively.

4.3 A Quaternary Hilbert Pattern

Chen et al. [36] proposes an approach for encoding $h = \mathcal{H}(x, y)$ and decoding $(x, y) = \mathcal{H}^{-1}(h)$ of the Hilbert curve. In the encoding procedure, there are given the coordinates of a particular point with a pair (x, y) the corresponding Hilbert value h is to be determined. Conversely, given the Hilbert value, h in the decoding procedure, the coordinates x and y are to be determined. The standard Hilbert curve with its Hilbert values h in the decimal representation for each coordinate x and y. The curve from one iteration n is replicated and moved onto the four quadrants of a larger square of the next iteration $n + 1$ with a suitable rotation. These four curves are joint by three line segments.

We define the upper left quadrant as quadrant 0, the upper right one as quadrant 1, the lower-right as quadrant 2, and the lower-left quadrant 3. By converting the decimal digits in Figure 4.3 into quaternary numbers in Figure 4.4 the highest quaternary digit also indicates the current number of the quadrant. In these Figures, we are using rotated and mirrored versions of the Hilbert curve compared to the previous sections' curve representation. We believe that this rotated version of the Hilbert curve fits the memory hierarchy in today's modern computer architecture best, since quadrant 1 profits from renewed use of the data by quadrant 0 and quadrant 3 profits from quadrant 2. Above all, the named quadrants are consecutive neighbors. Another regularity in the quaternary pattern is that quadrant $r = 0$ of iteration n (denoted as H_n) is a copy, reflected on the main diagonal, of the pattern of iteration $n - 1$ (H_{n-1}). Quadrants $r = 1$ and $r = 2$

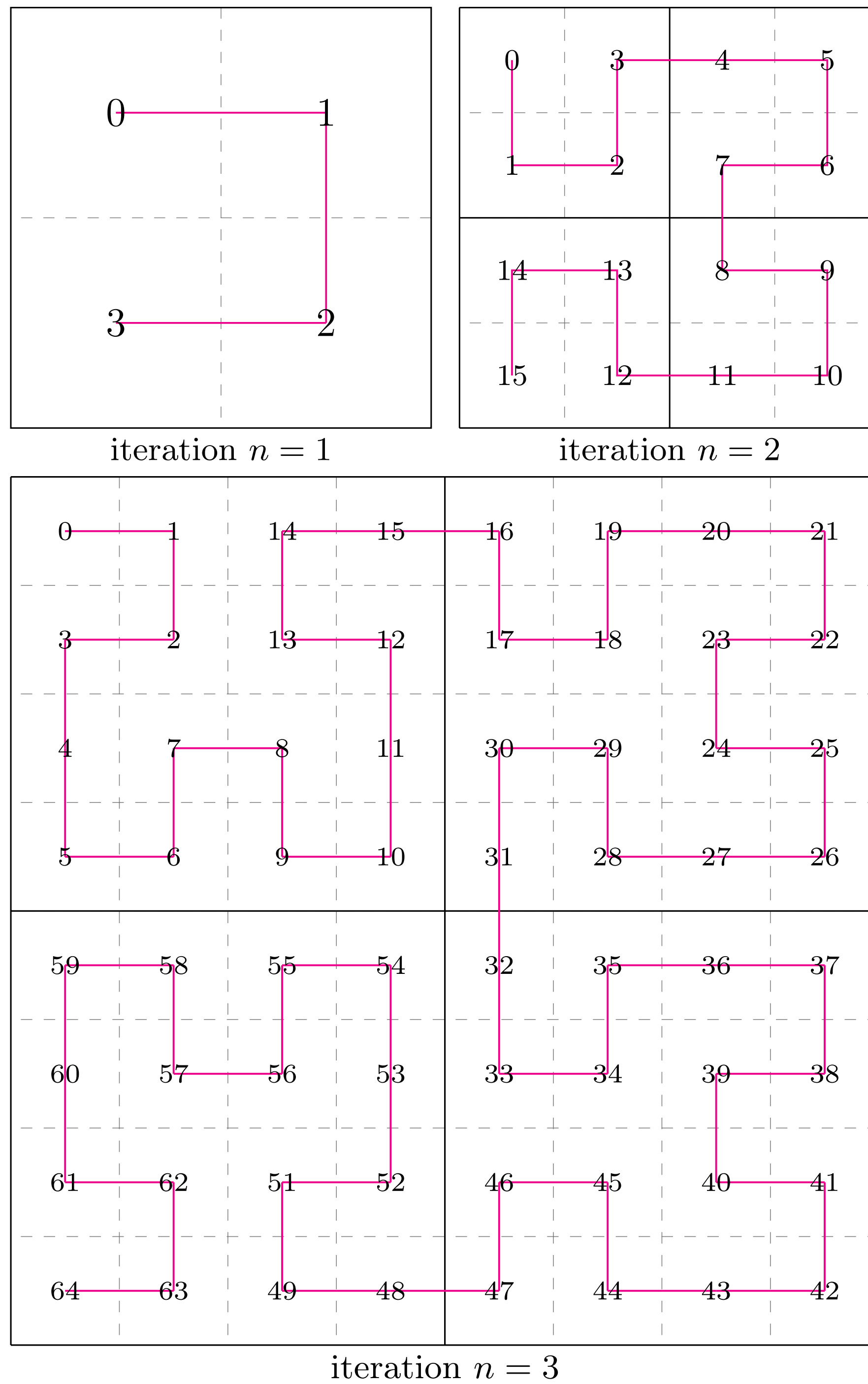


Figure 4.3: The Hilbert value h in decimal format for the first three iterations of the Hilbert curve.

of H_n are direct copies of H_{n-1} and quadrant $r = 3$ is a copy reflected on the minor diagonal. H_n would only have to be extended by the quadrant number for the leading quaternary digits.

For describing the encoding and decoding algorithm, we use the Cartesian coordinate system to express the positions of all the elements in the Hilbert matrix. In Figure 4.3, element $h = 0$ is the top-left element with coordinates $x = 0$ and $y = 0$. As an example, at iteration $n = 3$ the position of the cell with the Hilbert value 52 is $x = 3$ and $y = 6$. Both the encoding and the decoding algorithm consists of three steps: initialization, iteration, and decision.

Encoding algorithm, $h = \mathcal{H}(x, y)$

For the given coordinate pair x and y, we wish to derive the current Hilbert value h. At first we need to determine the current iteration number n, which is $n = \lfloor \log_2(max(x, y)) \rfloor + 1$ and the weight $w = 2^{n-1}$. After this initialization, an iteration step follows, where the quadrant r needs to be determined. The paper does not give more details on how they should be determined, but it seems to be a trivial step to determine on the basis of the Leitmotiv into which of the 4 quadrants the current coordinates reside. At next, r is concatenated to the Hilbert value h, the current coordinates x, y are then updated with respect to w based on a lookup table. The iteration variable n is decremented to repeat the procedure on a lower iteration level.

Decoding algorithm, $(x, y) = \mathcal{H}^{-1}(h)$

For a given iteration n, we wish to obtain the Cartesian coordinates for a Hilbert value h. The Hilbert value h is converted to a quaternary digit string, where the length of the string is the iteration level n. In the iteration step, the least significant digit of h determines how the pairs are updated. The variables are then updated according to a different table than the table proposed in the encoding step. Noteworthy mentioning is that they use different tables for even and odd iterations.

4.4 Observations

Based on the Hilbert curve in Figure 4.4, we discovered a pattern, which describes the transitions between individual states between the Hilbert value in quaternary format and

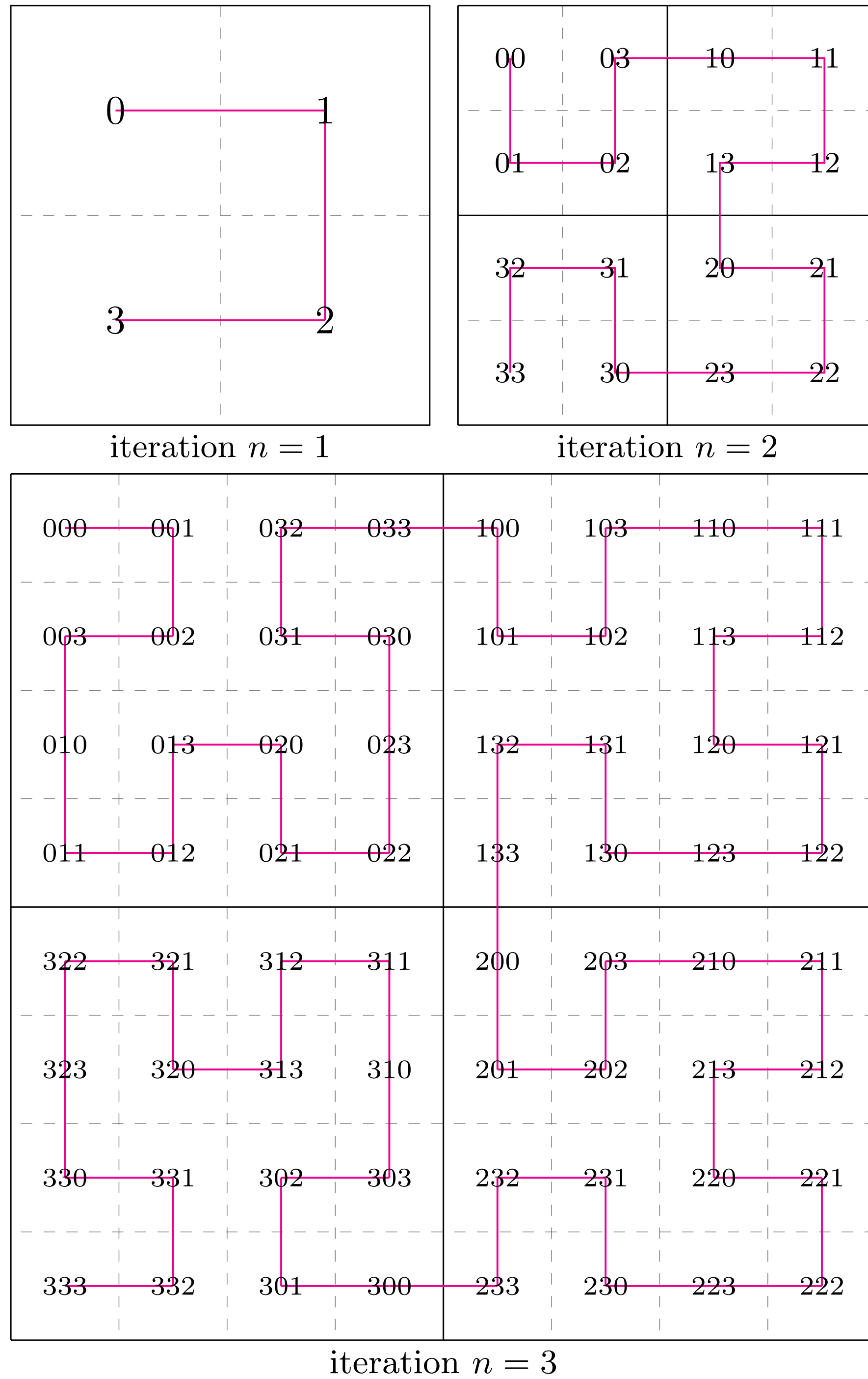


Figure 4.4: The Hilbert value h in the quaternary format for the first three iterations of the Hilbert curve.

the binary representation of the coordinates x and y. These coordinates can also be seen as iteration variables i and j from a loop:

for $h := 0$ **to** $n^2 - 1$ **do**

$(i, j) := \mathcal{H}^{-1}(h)$

process loop body(i, j)

There is also a connection between the variables i and j, and the Hilbert value h. First, the binary representation of the Hilbert value requires $2n$ bits, and the binary representation for one of the coordinates is required n bits. The algorithm from the paper [36] (see section 4.3) describes the encoding and decoding approach based on a quaternary Hilbert value. The quaternary Hilbert value contains already the quadrants which need to be visited down the iteration hierarchy. As an example we consider Figure 4.4 for iteration $n = 3$. The Hilbert value 52 in decimal format is in the quaternary format 303. This cell is located in quadrant 3. After dividing the third quadrant into 4 more quadrants, the cell is located in quadrant 0, and within quadrant 0, it is located in quadrant 3.

At next, we look at the orientation of the Hilbert value. The complete curve for $n = 3$ starts at quadrant 0, traverses through quadrant 1 and 2, and ends in quadrant 3. For abbreviation purposes, we denote this with the symbol $\boxed{\sqsupset}$. After dividing the top-level quadrant into sub-quadrants there are 3 different orientations for the Hilbert curve, namely $\boxed{\sqcup}$, two times the pattern $\boxed{\sqsupset}$ and $\boxed{\sqcap}$. The pattern $\boxed{\sqsubset}$ is missing. After deciding on one orientation pattern, we could write down one quaternary digit of the Hilbert curve and the next binary digit for the variables i and j. If we decide on one quaternary digit on the Hilbert curve as a state in a state machine, then the transition between states is equivalent to sub-dividing the matrices.

From the thoughts above, we can conclude for each state there are three different possible transitions. One transition, which is also very likely, is that the same orientation will be seen again. Therefore, in building a state machine for the Hilbert curve, it requires a recursive edge to the current state, and two other transitions are possible. For each transition, we can describe a bit shift on the variables i and j and two-bit shifts on the Hilbert value h. The resulting finite automation is explained as deterministic finite automation of a Mealy type in section 5.3.

4.5 L-systems

A Lindenmayer system (or L-system) is a parallel rewriting system and a type of generative grammars. Generative grammars have been first proposed by Noam Chomsky in the 1950 [39]. Such a grammar G is used to transform strings and mainly consists of a finite set of production rules (left-hand side $\leftarrow$ right-hand side). Each of these sides consists of the following symbols:

- A finite set N of *nonterminal symbols*, that is disjoint from the grammar G. A nonterminal symbol is represented by uppercase letters and they indicates that **some** production rule can be applied.
- A finite set Σ of *terminal symbols* that is disjoint from N and G. A terminal symbol is represented by symbols or lowercase letters. They indicate that **no** production rule can be applied.

L-systems are introduced and developed by Astrid Lindenmayer, a Hungarian theoretical biologist, and botanist in 1968 [81]. Lindenmayer used L-systems to describe the behaviour of plant cells and to model the growth processes of plant development. L-systems have also been used to model the morphology of a variety of organisms. Their recursive nature leads to self-similarity, and thereby they can be used to generate self-similar fractals or space-filling curves. Similarly to generative grammars, they have

- an alphabet Σ, containing all the terminal symbols.
- ω (start, axiom or initiaor) a string of symbols from Σ which defines an initial state of the system.
- N a set of production rules, defining how variables can be replaced with combinations of constants and other variables. They are similar to nonterminal symbols.

An L-system is called *context-free*, if it is based upon context-free grammar. A context-free grammar is a grammar where each production has the form $A \leftarrow w$, where A is a nonterminal, and w is a string of terminals and nonterminals. In a context-free grammar, any nonterminal can be expanded out to any of its productions at any point. Grammar is the set of strings of terminals that can be derived from the start symbol.

As a very simple example consider the following L-system:

variables (N): A,B
constants (Σ): none
axiom: A
rules: $(A \leftarrow AB)$, $(B \leftarrow A)$

which produces:

$n = 0:$ A
$n = 1:$ AB
$n = 2:$ ABA
$n = 3:$ ABAAB
...

L-systems can be used to generate fractal trees, the cantor set or the Sierpinski traingle. In section 5.3 we describe how to use the L-systems to describe the Hilbert curve.

4.6 Concluding Remarks

This chapter gave some general introduction into space-filling curves, their divers applicability, and an introductory example to L-systems. L-systems are a grammar that is used in our next chapter to describe the Hilbert curve without recursion.

Chapter 5

Cache-oblivious Hilbert Curve

In this chapter, we propose the Hilbert curve. A macro to replace canonical loops in algorithms to efficiently exploit the performance of modern cache hierarchies.

5.1 Introduction

In chapter 3 we introduced a cache-conscious [4] implementation of the K-means algorithm. For the loop traversal in this algorithm, we have chosen step-sizes on multiple loop levels (i.e., $4 \times 4 \times 4$ blocks, c.f. Figure 3.1), which are processed in every loop iteration independently from all other blocks. Provided that we have a single cache large enough to store blocks with a size of $4 \times 4 \times 4$, this strategy is dramatically better because we utilize the full bandwidth of our single cache. Modern processors support a memory hierarchy involving up to 3 levels of cache (L1, L2, L3), where the lowest level (L1) is the smallest, fastest, and the most expensive and the highest level (L3) the cheapest, largest, but slowest. While we might be able to determine the pure hardware size of all these cache mechanisms for a given hardware configuration, it is difficult to know (and subject to frequent changes) how much of the various caches are available for our matrices and not occupied, e.g., by other concurrent processes or the operating system.

To efficiently support the complete hierarchy of memories of (effectively) unknown sizes, we need a different concept: a **cache-oblivious algorithm** [55] is, unlike our above 3-loop construct, not optimized for single, known cache size. It follows a strategy supporting a wide range of different cache sizes, which can also be present simultaneously. The idea is to systematically interchange the increment of the variables i and j such that the locality of the accesses to both types of objects (i **and** j) is guaranteed. Space-filling

curves like the Hilbert curve or the Z-order curve act to some degree like the nesting of a high number ($2 \cdot \log_2 n$) of loops going forward and backward with different step-sizes.

5.2 Locality of the Hilbert Curve

In Figure 5.1 we can recognize (a) the cyclic access pattern of nested loops, (b) the cache-oblivious access pattern of the Hilbert curve, (c) the histories of variable i and (d) j over time, and (e) the number of cache misses over varying cache size. We can see in Fig. 5.1(d) that the access pattern of the variable j yields much more locality for the Hilbert loops compared to the cyclic access pattern of the nested loops. The result (e) is a dramatically improved number of cache misses, particularly for realistic cache sizes, like 5-20% of the main memory.

5.3 Well-known Methods for the Hilbert Curve

For an introduction to the Hilbert curve, its historical context, and construction approaches, we refer the interested reader to chapter 4. Here, we introduce the well-known approaches in this section following an automaton-theoretic point of view. Many iterative approaches to generate space-filling curves like [36] can be regarded as a deterministic finite automaton, and many recursive approaches like [27] as a context-free grammar.

Explicit Enumeration of Hilbert Values

The simplest way of generating a loop over the two variables i and j enumerating the pairs in Hilbert-order (or any other space-filling curve) is to iterate over all possible Hilbert values h and to apply the inverse Hilbert function $\mathcal{H}^{-1}(h)$. Let us for this section assume that both i and j iterate over the range $0 \le i, j < n$ where n is a power of two:

for $h := 0$ **to** $n^2 - 1$ **do**
  $(i, j) := \mathcal{H}^{-1}(h)$;
  process object pair (i, j);

For matrix multiplication, we substitute our placeholder:

$$\textit{process object pair } (i,j) \quad \Leftarrow \quad a_{i,j} := \textstyle\sum_k \, b_{i,k} \cdot c^{\mathsf{T}}_{j,k}.$$

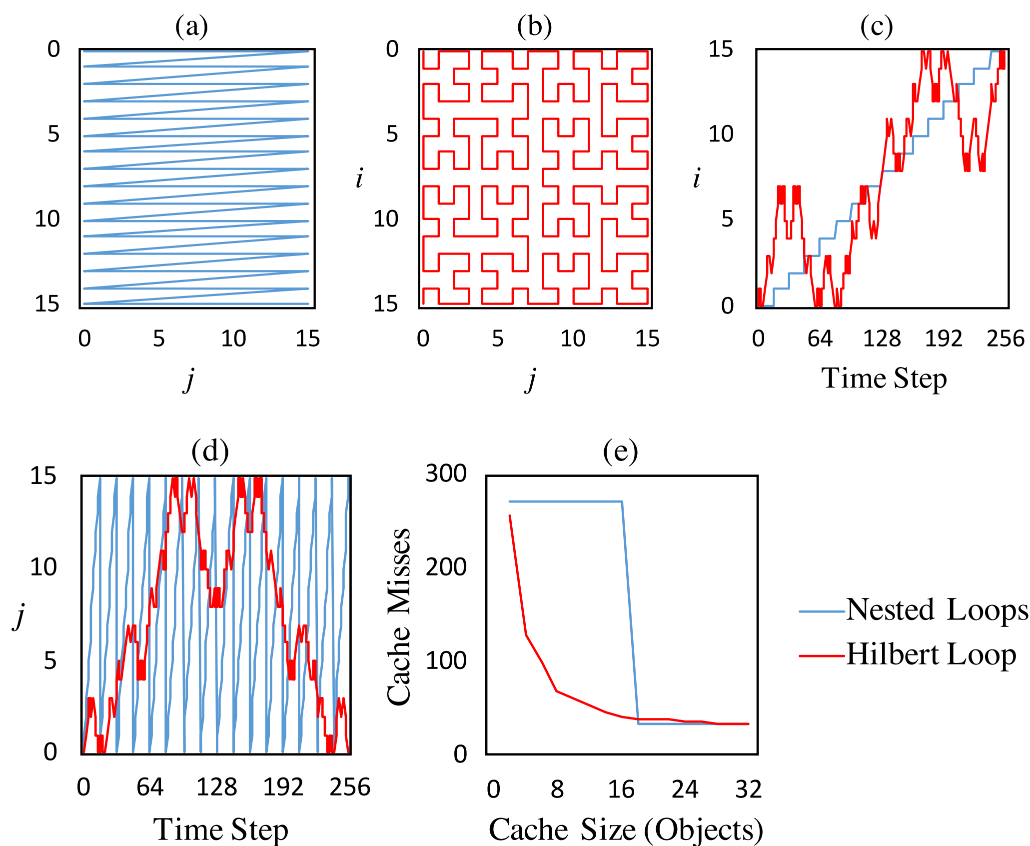


Figure 5.1: Comparison of the Traversal Order for Nested Loops (a) and Hilbert Loops (b). An improved locality can be recognized in the histories over time for variable i (c) and j (d), and a considerably improved cache miss rate (e).

The inverse Hilbert function is depicted in Figure 5.2 as a deterministic finite automaton (DFA) of Mealy-type (i.e., the output, the bit-strings representing i and j are generated at the state transitions of the DFA). The bit-string representation of the Hilbert value h, divided into groups of 2 bits, is the input of the DFA. State 3 is used as a starting state to generate the Hilbert curve in the clockwise order. Alternatively, state 2 can also be used to start to generate an anti-clockwise curve.

Appending a binary digit 1 to the output bit-string i (in symbols: $i \twoheadleftarrow 1$) represents the mathematical operation $i := 2 \cdot i + 1$. With the example input $h = 52_{10} = 110100_2$, our DFA makes upon the first bit-pair 11_2 of the input string the first transition from

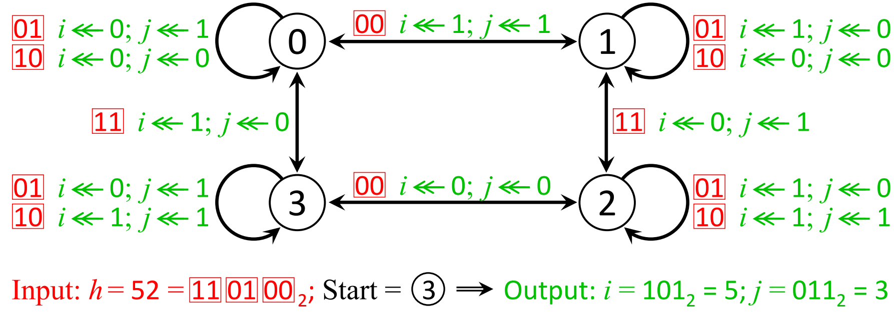


Figure 5.2: Mealy-DFA for Inverse Hilbert: $(i, j) := \mathcal{H}^{-1}(h)$ to generate variables i and j from the Hilbert value h.

State 3 to State 0, then for the second bit-pair 01_2 stays in State 0 and goes finally with bit-pair 00 to State 1. In these three state transitions it appends 1, 0, and 1 to i; 0, 1, and 1 to j to finally obtain $(i, j) = (5, 3)$ in decimal system. Each of the $\log_2 n$ bit-pairs of h is separately processed by the DFA. In contrast to this $O(\log n)$ overhead, a pair of two nested loops have only a constant overhead per loop iteration (increment i and/or j). The overhead could be prohibitive for many applications.

When the upper limits of the loops are different $(0 \leq i < n, 0 \leq j < m)$ or do not correspond to a power of two, then we have two options: either we generate a bigger Hilbert curve and suppress the processing of pairs (i, j) that are actually out of the range:

for $h := 0$ **to** $2^{2\cdot\lceil\log_2 \max\{n,m\}\rceil} - 1$ **do**
    $(i, j) := \mathcal{H}^{-1}(h)$;
    if $i < n$ **and** $j < m$ **then**
        process object pair (i, j);

or we generate a Hilbert curve of side length $2^{\lfloor\log_2 \min\{n,m\}\rfloor}$ and complement the missing values later in additional loops with smaller Hilbert curves [40]. Both options incur a high overhead and deteriorate the goal of cache-obliviousness.

Lindenmayer-Systems

In section 4.5 we give a brief introduction into the basics of Lindenmayer systems and context free grammar.

For subsequent calls $\mathcal{H}^{-1}(h), \mathcal{H}^{-1}(h+1)$, it is likely that the bit-strings representing h and $h+1$ have a long common prefix for which the DFA makes the same state transitions. Moreover, by the Hilbert curve's properties, it is guaranteed that the corresponding coordinates (i, j) generated by subsequent Hilbert values differ exactly by 1. An alternative way to define an iteration overall values of two variables in Hilbert order avoiding this unnecessary workload is the Lindenmayer system:

Definition 1 (Lindenmayer-System for the Hilbert-Curve)**.** Let A, B be the nonterminal symbols and $\oplus, \ominus, \triangleright, \pi$ the terminal symbols of a context-free grammar *(CFG)* involving the following production rules:

$$
\begin{array}{lcl}
A & \rightarrow & \pi \mid \ominus\ B \triangleright \oplus\ A \triangleright A \oplus \triangleright B \ominus \\
B & \rightarrow & \pi \mid \oplus\ A \triangleright \ominus\ B \triangleright B \ominus \triangleright A \oplus
\end{array}
$$

The terminal symbols represent the graphical operations:

$\ominus$ turn 90 degrees to the left without moving,

$\oplus$ turn 90 degrees to the right without moving,

$\triangleright$ go forward one step in the current direction d,

π process object pair (i, j),

on a grid of size $n \times n$ where n is a power of two, oriented such that j is drawn from left to right and i from top down.

Direction: The coding and semantics of d is as follows:

$$
\begin{array}{ll}
d = 0 \Leftrightarrow \text{look left:} & \text{the next } \triangleright\text{-step will do } j := j - 1, \\
d = 1 \Leftrightarrow \text{look up:} & \text{the next } \triangleright\text{-step will do } i := i - 1, \\
d = 2 \Leftrightarrow \text{look right:} & \text{the next } \triangleright\text{-step will do } j := j + 1, \\
d = 3 \Leftrightarrow \text{look down:} & \text{the next } \triangleright\text{-step will do } i := i + 1.
\end{array}
$$

Axiom (Start Symbol): We use ***either*** A with initialization $d = 3$ ***or*** B with initialization $d = 2$.

Level ℓ of a rule expansion: The expansion of the axiom has level $\ell = \log_2 n$. If a nonterminal symbol appears on the right side of a production rule of level ℓ, its expansion has level $\ell - 1$. The terminating productions $A \rightarrow \pi$ and $B \rightarrow \pi$ are applied ***exactly*** at level $\ell = 0$.

While the Mealy-DFA of Section 5.3 generates only one (i, j)-pair, the Lindenmayer-CFG produces the whole $n \times n$ Hilbert curve when we start at level $\ell = \log_2 n$. Axiom A with $d = 3$ generates the values in a clockwise order starting from $(i, j) = (0, 0)$ and ending at $(n, 0)$, and axiom B with $d = 2$ in an anticlockwise order ending at $(0, n)$.

According to the definition of d, the operations $\oplus$ and $\ominus$ correspond to the cyclic increment/decrement of d:

$$\begin{aligned} d &:= (d+1) \ \mathbf{mod}\ 4; \quad // \ \oplus \\ d &:= (d+3) \ \mathbf{mod}\ 4; \quad // \ \ominus \end{aligned}$$

To avoid expensive (pipeline-breaking) **if**-**else**-operations, we use the following implementation of the forward-step:

$$\left.\begin{aligned} j &:= j + ((d-1) \ \mathbf{mod}\ 2); \\ i &:= i + ((d-2) \ \mathbf{mod}\ 2); \\ h &:= h + 1; \end{aligned}\right\} \quad // \ \triangleright$$

The **mod**-operation preserves the sign. For $d = \langle 0, ..., 3\rangle$ we get $i := i + \langle 0, -1, 0, +1\rangle$ and $j := j + \langle -1, 0, +1, 0\rangle$. variable (either i or j) is truly affected.

The Lindenmayer system to produce the whole sequence of Hilbert values can be straightforward implemented with two recursive functions $A(\ell)$ and $B(\ell)$, exactly performing the above operations on global variables $h, i, j, d \in \mathbb{N}_0$:

Analogously **function** $B(\ell)$; the labels are not needed for the implementation but for the following analysis. Note the comments are giving the corresponding terminal and nonterminal symbols from the context-free grammar. Although all the increase and decrease operations corresponding to the commands $\triangleright$, $\oplus$, and $\ominus$ have now constant complexity, the recursive implementation still has some drawbacks: Firstly, after the generation of 4 subsequent (i, j)-pairs, one incarnation of $A(\ell)$ or $B(\ell)$ is finished. We

Algorithm 1 Recursive Lindenmayer Algorithm.

```
1          function A(ℓ)
2              if ℓ = 0 then
3                  process object pair (i, j);          // π
4              else
5                   d := (d + 3) mod 4;                   // ⊖
6       labelA0:    B(ℓ − 1);                             // B
7                   j  := j + ((d − 1) mod 2);  ⎫
8                   i  := i + ((d − 2) mod 2);  ⎬         // ▷
9                   h  := h + 1;                ⎭
10                  d := (d + 1) mod 4;                   // ⊕
11      labelA1:    A(ℓ − 1);                             // A
12                  j  := j + ((d − 1) mod 2);  ⎫
13                  i  := i + ((d − 2) mod 2);  ⎬         // ▷
14                  h  := h + 1;                ⎭
15      labelA2:    A(ℓ − 1);                             // A
16                  d := (d + 1) mod 4;                   // ⊕
17                  j  := j + ((d − 1) mod 2);  ⎫
18                  i  := i + ((d − 2) mod 2);  ⎬         // ▷
19                  h  := h + 1;                ⎭
20      labelA3:    B(ℓ − 1);                             // B
21                  d := (d + 3) mod 4;                   // ⊖
```

have to return to one of labelA$_{0..3}$ or labelB$_{0..3}$, perform the next actions and then start the next recursive calls. In summary, after every 4^k iterations, we have to move up and down on the stack at least for k positions where $1 \leq k \leq \log_2 n$. This is still a logarithmic overhead per loop iteration in a worst-case-analysis (but since the worst case does not occur frequently, it is indeed constant in the average-case when applying amortized analysis). Secondly, grids with different side lengths not corresponding to the powers of two are still open in the recursive solution.

Thirdly, the Lindenmayer system's recursive nature is an obstacle to the implementation and compiler-optimization of the host algorithm (like matrix multiplication, etc.). The core of a host algorithm must be implemented twice in the terminating cases of the functions $A(\ell)$ and $B(\ell)$, which can communicate only via global variables. More importantly, the compiler has fewer options for optimization. In C-like languages, optimization is only done inside functions, not across function calls. Therefore, it is advisable to put some effort in making our Lindenmayer system non-recursive, as described in the following section.

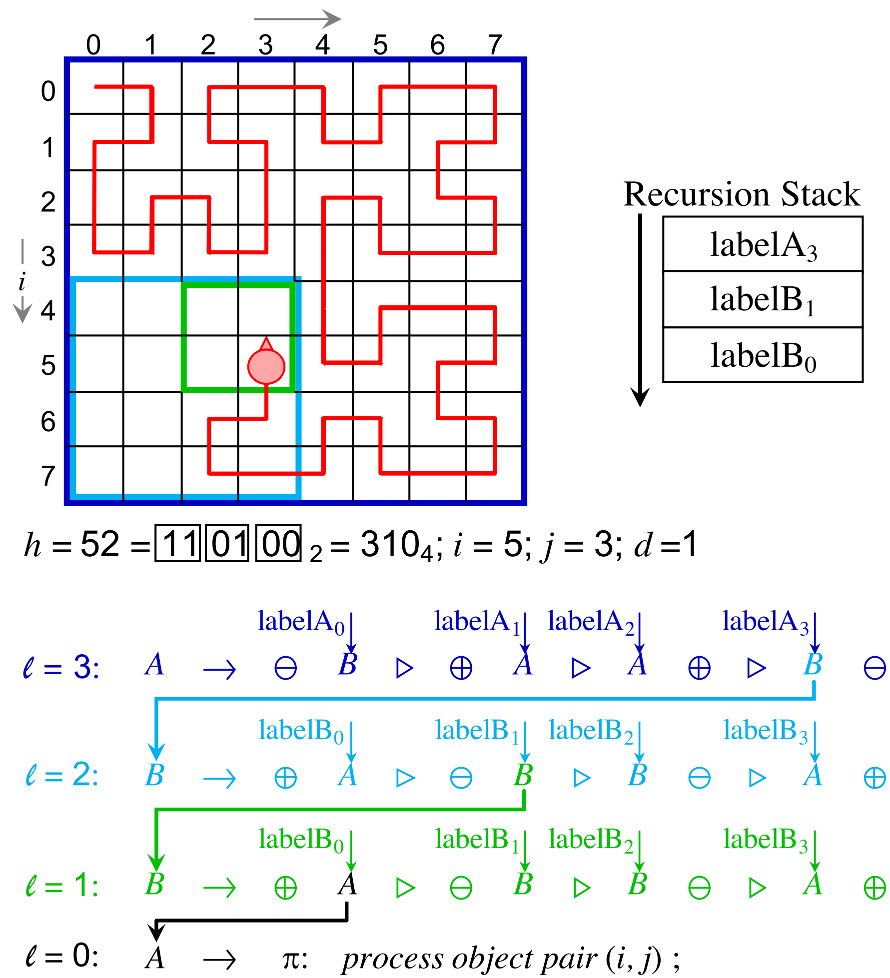


Figure 5.3: Recursive generation of (i, j)-pairs following the Hilbert-curve.

5.4 Novel Non-recursive Lindenmayer

The functions $A(\ell)$ and $B(\ell)$ are not straightforward to make iterative. In regular time intervals, we have to leave one or more recursive incarnations of $A(\ell)$ or $B(\ell)$, return to the middle of another A or B-function on the recursion stack, perform the next action of this incarnation then, and start new recursive calls again. This can be studied in Figure 5.3, where we are in the middle of the generation of the Hilbert loop, at $i = 5, j = 3, h = 52$. At this point, we have four active incarnations, one of production rule A, two of B, and the terminating rule $A \to \pi$. The dark blue printed rule at $\ell = 3 = \log_2 n$ generates the Hilbert curve of the whole (i, j)-grid (surrounded by a dark blue frame). We are

currently in the last (lower left) sub-quadrant, marked by a light blue frame, and the corresponding position in production rule A is labelA$_3$, which is expanded in the next, light blue production rule. In a recursive implementation, we have labelA$_3$ on the stack. The production rule $B(\ell = 2)$, in turn, is expanded at labelB$_1$ (next position on the recursion stack) corresponding to the green production rule (again B) and the green frame in the grid. When ending the green production rule, we have to return to labelB$_1$ of the light blue rule ($\ell = 2$), perform the corresponding action ($\triangleright$, between labelB$_1$ and labelB$_2$), and then make the next recursive call $B(\ell - 1)$ at labelB$_2$. The labels which are on the stack, labelA$_{0..3}$, labelB$_{0..3}$ agree with the definitions in **function** $A(\ell)$ and $B(\ell)$. Where appropriate, we will also note e.g. labelX$_0$, meaning: labelA$_0$ *or* labelB$_0$.

Our idea to make the recursive algorithm iterative is to code the complete recursion stack in a single $\mathbb{N}_0$-variable (e.g., 64 bit, possibly a register of the CPU). We will demonstrate that we do not need a new variable for this purpose. Instead, the current Hilbert value h, as well as the current direction code d, contain all the information to derive both the current recursion depth, as well as (for all recursive incarnations of the methods A and B) the current position where we have to return. We split the proof of this into two parts: Lemma 1 states that the number of each label on the recursion stack exactly corresponds to the Hilbert-value grouped in bit-pairs. This is already obvious from Figure 5.3 where the Hilbert value $h = 52 = 11\ 01\ 00_2 = 310_4$ noted in the 4-adic system is identical to the numbers on the stack: (labelA$_3$, labelB$_1$, labelB$_0$). Lemma 2 shows how to derive the information whether we are in $A(\ell)$ or $B(\ell)$.

Lemma 1 (Label-Number)**.**

Consider an incarnation of the non-terminating rules:

$$\begin{array}{lllr} A & \rightarrow & \ominus\ B \triangleright \oplus\ A \triangleright A\ \oplus \triangleright B\ \ominus & \text{or} \\ B & \rightarrow & \oplus\ A \triangleright \ominus\ B \triangleright B\ \ominus \triangleright A\ \oplus & \text{at level } \ell. \end{array}$$

(1) The number $\mathrm{pop}(\ell)$ of processed object pairs (i, j) starting from the incarnation at level ℓ is $\mathrm{pop}(\ell) = 4^\ell$.

(2) The incarnation overall increases h by $\mathrm{forw}(\ell) = 4^\ell - 1$.

(3) For each Hilbert value h generated at *labelX$_k$*, the value

$$a := \lfloor h/4^{\ell-1} \rfloor \ \mathbf{mod}\ 4 \quad \textit{equals the label number} \quad k.$$

Proof. (1) At the bottom level $\ell = 0$, the number of processed object pairs is $pop(\ell) = 1$. For $\ell \geq 1$, the number of expansions to π is multiplied by four for each level: $pop(\ell) = 4 \cdot pop(\ell - 1)$. Consequently, $pop(\ell) = 4^\ell$.

(2) At the bottom level, we have a number of forward steps $forw(\ell) = 0$. In each higher level, we have four times as many forward steps as in level $\ell - 1$ plus additional three performed in the current grammar rule:

$$\forall \ell \geq 1 : forw(\ell) = 4 \cdot forw(\ell - 1) + 3 \Rightarrow forw(\ell) = 4^\ell - 1.$$

(3) At the beginning $k = 0$ of our rule, it is possible that rules of the same level have been processed before. If so, they must have been completely processed. According to (2), each of these rules have increased h by $forw(\ell) = 4^\ell - 1$, and together with the final $\triangleright$ from the parent rule, we have some multiple $r \cdot 4^\ell, (\exists r \in \mathbb{N}_0)$. For $k = 0$ we have $a = \lfloor r \cdot 4^\ell / 4^{\ell-1} \rfloor \textbf{ mod } 4 = (r \cdot 4) \textbf{ mod } 4 = 0$.

At position $k = 1$, compared to $k = 0$, we have increased h by $4^{\ell-1}$, because we have applied one rule at level $\ell - 1$ (cf. (2) again) and performed an additional $\triangleright$-step. Thus, we have $a = \lfloor (r \cdot 4^\ell + 4^{\ell-1}) / 4^{\ell-1} \rfloor \textbf{ mod } 4 = (4r + 1) \textbf{ mod } 4 = 1$. In the general case $k \in \{0, ..., 3\}$ we do this k times: $a = \lfloor (r \cdot 4^\ell + k \cdot 4^{\ell-1}) / 4^{\ell-1} \rfloor \textbf{ mod } 4 = k$. □

Next we show how to decide according to h and d whether we are in grammar rule A or B, i.e. the *letter* of the labels.

Lemma 2 (Production Rule A or B)**.**

(1) The direction code d is the same at the beginning and at the end of a production rule.

(2) At the beginning and end of the production rule A, the parity of the direction code d is always odd, at the beginning and end of B always even.

(3) The parity of d combined with the position (e.g. identified by the label) decides if we are in grammar rule A or B.

Proof. (1) Proof by structural induction over the production rules: In the **base clauses**, $A \rightarrow \pi$ and $B \rightarrow \pi$ the direction code is not changed, thus (1) is trivially true.

Induction step: Consider the production rule:

$$A \quad \rightarrow \quad \ominus\ B\ \triangleright\ \oplus\ A\ \triangleright\ A\ \oplus\ \triangleright\ B\ \ominus$$

If the direction code is not changed in the expansion of the nonterminals on the right-hand side, then it is not changed in the application of A because the number of $\oplus$ is the same as the number of $\ominus$ (two). The same is true for rule B.

(2) By structural induction: **Base clauses**: At the beginning of the first expansion of the axiom the statement is trivially true because according to Def. 1 we either start with A and initialize $d = 3$ (odd) or start with B and initialize $d = 2$ (even). Because of (1) this is also true at the end of the first (topmost) expansion of A or B.
Induction step: Consider again

$$A \ \rightarrow\ \overset{\text{labelA}_0\downarrow}{\ominus}\ B\ \triangleright\ \overset{\text{labelA}_1\downarrow}{\oplus}\ A\ \overset{\text{labelA}_2\downarrow}{\triangleright}\ A\ \oplus\ \overset{\text{labelA}_3\downarrow}{\triangleright}\ B\ \ominus$$

Before expanding B at labelA$_0$ we have applied $\ominus$. Note that both operators $\oplus$ and $\ominus$ toggle the parity of d from even to odd and vice versa because they add an odd number (1 or 3) to d (**mod** 4). Thus, if d is odd at the start of expansion A (induction hypothesis), it is even at the start of expansion B, odd again at A at labelA$_{1..2}$, and even at B at labelA$_3$. Analogously in the expansion of B: if d is even at the beginning (induction hypothesis), d is odd at every nonterminal A and even at every B. This is a complete analysis of all cases, and to summarize the induction step: if (2) is true for the nonterminal on the left side of a rule (at level ℓ), it must be true for all nonterminals on the right side and thus for all left sides of the next level ($\ell - 1$).

(3) As a consequence of (2), we know the parity at the beginning of a rule expansion. Since the parity is switched at each $\oplus$ and $\ominus$, we can mark the parity of d for all positions:

$$A \ \rightarrow\ \ominus\ B\ \triangleright\ \oplus\ A\ \triangleright\ A\ \oplus\ \triangleright\ B\ \ominus$$

odd | even | odd | even | odd

$$B \ \rightarrow\ \oplus\ A\ \triangleright\ \ominus\ B\ \triangleright\ B\ \ominus\ \triangleright\ A\ \oplus$$

even | odd | even | odd | even

We can see that the parity differs between A and B at every position. Therefore,

the combination of position and parity decides the rule A or B in which we currently are. □

Now, Lemma 1 and 2 would enable us to exactly mimic the recursion with its stack. However, we would still have the logarithmic worst-case complexity. Instead, we further transform our algorithm into that in Figure 2 which truly performs a loop enumerating the Hilbert values h. In its body, we perform the operation π (process (i, j)) and, then decide where we would be in the recursive system and which action a must be executed. An action takes always place between two successive recursive calls and has the label of the latter (e.g. $a = 2$ is between labelX$_1$ and labelX$_2$), and corresponds always to a forward-step, potentially preceded or followed by a $\oplus$ or $\ominus$-step. E.g. in production rule A at $a = 1$ we perform $\oplus \triangleright$, and at $a = 3$ we perform $\triangleright \oplus$ etc. As there is no forward-step before labelX$_0$ and after labelX$_3$ we only consider the three actions $a \in \{1, 2, 3\}$, but not 0. We will see later how to cope with the $\oplus$ and $\ominus$ at the beginning and end of the rules.

To obtain the right action code, we first increase h. If we are at the end of one or more production rules on the stack, then increasing h will change one or more 11_2-bit-pairs at the end of h into 00_2. (cf. Lemma 1). In this case, the first bit-pair (from right to left) $\neq 00_2$ defines the level ℓ and the action a to be performed. Therefore, we determine (after increasing h) the number of 00_2-bit-pairs at the end in constant time by the following trick: If the bitwise and-operation is applied to h and its negative complement $-h$, then in the result all bits are 0 up to one exception: The last (least significant) 1 which has been set in h is still set (the result is the largest power of two which divides h with no rest). The binary logarithm $\lfloor \frac{1}{2} \log_2(h \; \mathbf{and}_{\text{bitw}} -h) \rfloor$ corresponds to the number of zero-

Algorithm 2 The Non-recursive Lindenmayer Alg.

```
 1 function LindenmayerNonRecursive()
 2     (i, j) := (0, 0); h := 0; d := 3;
 3     while h < n^2 do
 4         process object pair (i, j);
 5         h  := h + 1;
 6         ℓ  := ⌊½ log_2(h and_bitw −h)⌋ + 1;
 7         a  := ⌊h/4^(ℓ−1)⌋ mod 4;
 8         d  := d xor_bitw (11_2 · (isOdd(ℓ−1) xor a = 3));
 9         j  := j + ((d − 1) mod 2);
10         i  := i + ((d − 2) mod 2);
11         d  := d xor_bitw (isOdd(ℓ−1) xor a = 1);
```

pairs at the end of h and equals $\ell - 1$. We determine the binary logarithm by casting the result of the bitwise **and** to a double-precision floating point number and extracting the exponent, which is very efficient and works for $1 \leq h \leq 2^{52} - 1$, the greatest natural number that can be represented by double-precision floating point numbers (according to IEEE-754) at no loss of precision. The action code then is extracted from h using

$$a := \lfloor h/4^{\ell-1} \rfloor \ \mathbf{mod}\ 4, \qquad \text{cf. Lemma 1.}$$

Finally, we perform the action in a by modifying d, in-/decreasing i or j, and modifying d again. The two modifications of d subsume the different $\oplus$ and $\ominus$-operations which are defined between two labels. Let us first assume we are at level $\ell = 1$, i.e. we do not have to consider additional $\oplus, \ominus$-operations from starting or ending recursive calls. As we know that even and odd direction codes can only be present at certain positions of A and B (e.g. at labelA_0, d is even, at labelB_0, d is odd, cf. Lemma 2), we can construct the lookup table (see Table 5.1) for the result d^{new} of the $\oplus$ or $\ominus$ operation *before* and *after* the forward step $\triangleright$:

$\oplus/\ominus$ before $\triangleright$ (ℓ odd)				$\oplus/\ominus$ after $\triangleright$ (ℓ odd)			
d^{old}	$a=1$	$a=2$	$a=3$	d^{old}	$a=1$	$a=2$	$a=3$
0	–	–	3	0	1	–	–
1	–	–	2	1	0	–	–
2	–	–	1	2	3	–	–
3	–	–	0	3	2	–	–

Table 5.1: Lookup table to derive the direction code d for the odd cases of ℓ.

Here, "–" means $d^{\mathsf{new}} = d^{\mathsf{old}}$ (no change) and is used for better visibility. Colors indicate if the transition from d^{old} to d^{new} has been caused by grammar rule A (blue) or B (green), as suggested by the parity of d^{old}. In the actions before $\triangleright$, both bits of d are reverted ($d := d\ \mathbf{xor}_{\text{bitw}}\ 11_2$) whenever $a = 3$. After $\triangleright$, the lower bit of d is reverted ($d := d\ \mathbf{xor}_{\text{bitw}}\ 01_2$) whenever $a = 1$. The $\oplus$ and $\ominus$ operations at the begin and end of production rules have the following influence: if ℓ is odd, we have an additional even number of $\oplus$ and $\ominus$. The parity of d does not change and the table above is still valid. If ℓ is even, the parity of d toggles before and after $\triangleright$ and the table changes into:

Overall we can express the action before and after $\triangleright$ by the bitwise logical operations shown in the algorithm in Figure 2 where (as in C-like languages usually) results of boolean operations (isOdd, "=", **xor**) are represented by 1 or 0 and can thus e.g. be multiplied with other values. The operations "$\ldots+1$" in Line 6 and "$\ldots-1$" in Line 7,8,11

$\oplus/\ominus$ before $\triangleright$ (ℓ even)				$\oplus/\ominus$ after $\triangleright$ (ℓ even)			
d^{old}	$a=1$	$a=2$	$a=3$	d^{old}	$a=1$	$a=2$	$a=3$
0	3	3	–	0	–	1	1
1	2	2	–	1	–	0	0
2	1	1	–	2	–	3	3
3	0	0	–	3	–	2	2

Table 5.2: Lookup table to derive the direction code d for the even cases of ℓ.

can be omitted or are removed by the optimizer. They are included in the pseudo-code for equivalence with the functions $A(\ell), B(\ell)$.

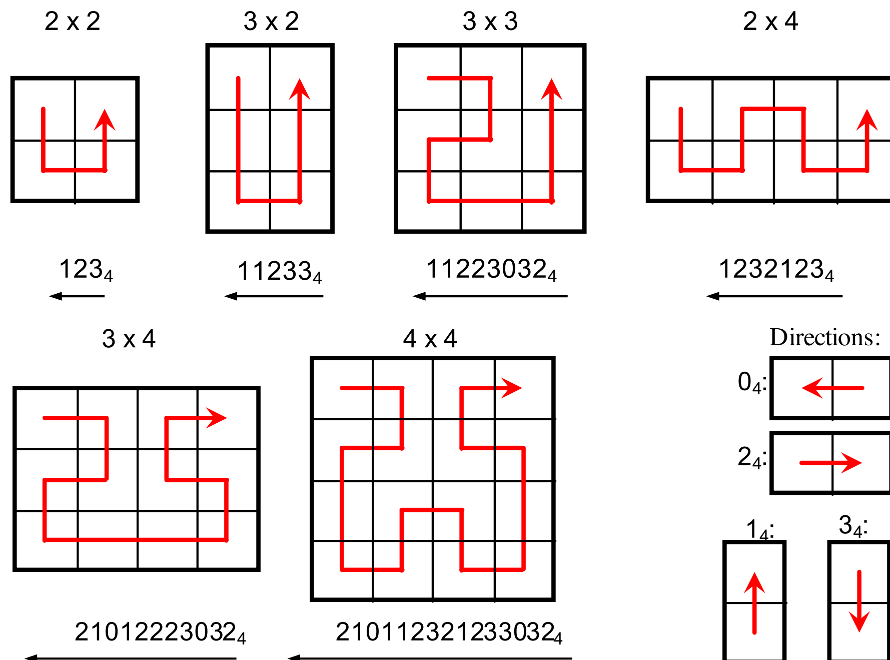


Figure 5.4: Examples of Nano-programs for Grids Ranging from 2×2 to 4×4 (all having basic orientation $d = 2$).

Lemma 3 (Complexity of LindenmayerNonRecursive)**.** The worst-case time complexity of our algorithm is constant per loop iteration.

Proof. The while-loop (lines 5–11) of our algorithm contains only elementary operations ($+$, **and**, **mod**, etc.). The number of these operations is constant (24). □

5.5 Nano-Programs

We back up our nonrecursive implementation of cache-oblivious loops by a concept called nano-programs. Nano-programs are small pieces of pre-computed space-filling curves for grid sizes $r \times s = 2\times 2, 2\times 3, 2\times 4, 3\times 4$, or 4×4 (see Figure 5.4). Degenerate grids with size $1 \times \{1, 2, 3, 4\}$ (single loop) and $0 \times \{0, 1, 2, 3, 4\}$ (empty loop) are also possible but are used only if either i or j overall has these degenerate limits. Nano-programs serve a two-fold purpose: They further decrease the overhead compared to the method proposed in the previous section because the management of helping variables like the direction d is simplified for the pre-computed small grids. The overall number of basic operations (additions, multiplications etc.) per loop iteration is thus reduced from 24 to 9. The second and more important purpose of our nano-programs is to enable grid-sizes which do not precisely correspond to a power of two, because as we show in the following, it is possible to tesselate a grid of any size $n \times m$ by a number of sub-grids each having size $r \times s = \{2, 3, 4\} \times \{2, 3, 4\}$. Let us start with the case that n and m are possibly different but in the same power of two: $t := \lfloor \log_2 n \rfloor = \lfloor \log_2 m \rfloor$. We can partition the $n \times m$ grid into a number $2^{t-1} \times 2^{t-1}$ of sub-grids where the height r is always an integer close to the *average* sub-grid height $\tilde{r} = n/2^{\lfloor \log_2 n \rfloor - 1}$. The following equation shows that r is always between 2 and 4:

$$\tilde{r} = n/2^{t-1} = n/2^{\lfloor \log_2 n \rfloor - 1} \begin{cases} \geq n/2^{(\log_2 n)-1} = 2, \\ \leq n/2^{(\log_2 n)-2} = 4, \end{cases}$$

Sub-grids of height $r = 4$ can also occur: e.g. if $n/2^{t-1} = 3.5$ then half of the sub-grids are of height 3 and 4. Analogously the width s: $2 \leq s = m/2^{t-1} < 4$. The sub-grids all have size $\{2, 3, 4\} \times \{2, 3, 4\}$.

A nano-program is a bit-sequence that can be stored in an integer variable P, which may be assigned to a register by the compiler while working with it. The bit-sequence contains codes similar to the direction-codes stored in the variable d (cf. Section 5.3). The nano-programs read from right to left, and therefore, the first operation to be performed on the variables i and j can be extracted from P and stored into a temporary variable c by $c := P \textbf{ mod } 4$. The current operation is then executed (i and j are updated based on c as in Section 5.3) and removed from P by a right-shift $P := \lfloor P/4 \rfloor$ until the nano-program is empty. To each sub-grid cell of size $r \times s$ there belongs a nano-program of size $r \cdot s - 1$ digits from a 4-ary system, i.e. $(r \cdot s - 1) \cdot 2$ bit. We have separate nano-programs for every size of the basis-grid $\{0, 1, 2, 3, 4\} \times \{0, 1, 2, 3, 4\}$ and for every orientation defined by the

variable d, totalling in a number of $5 \cdot 5 \cdot 4 = 100$ nano-programs of sizes up to 30 bit (integer register). A few examples of nano-programs (all for orientation $d = 2$; patterns for other orientations obtained by rotation) are visualized in Figure 5.4: we can see the graphical access patterns for $2 \times 2 ... 4 \times 4$ grids and the corresponding nano-programs. Note that these programs read from right to left, and they are here noted in a 4-ary system. The first movement of the 2×2 pattern (step down) is coded by the tailing digit 3 of the nano-program $123_4 = 27_{10}$.

The pseudo-code is embedded in the algorithm LindenmayerNonRecursive() and all processing of nano-programs highlighted in Algorithm 3. The size $(r \times s)$ of each cell of the nano-program is determined in Line 3 such that the sub-grid heights $r = 2, 3$, and 4 are evenly distributed to sum up to n (analogously for the widths s). After a complete nano-program consisting of $r \cdot s - 1$ steps has been processed, a final movement (lines 11–16) is performed to connect the previous sub-grid cell to the subsequent one.

Algorithm 3 Lindenmayer with Nano-programs.

```
1  (i,j) := (I,J) := (0,0); h := 0; d := 3; t := ⌊log_2 n⌋;
2  while h < 2^(2t−2) do
3      r := ⌊(I+1)·n / t⌋ − ⌊I·n / t⌋; s := ⌊(J+1)·m / t⌋ − ⌊J·m / t⌋;
4      P := nanoprograms[r][s][d];
5      while P ≠ 0 do
6          process object pair (i,j);
7          c := P mod 4;
8          P := ⌊P/4⌋;
9          j := j + ((c − 1) mod 2);
10         i := i − ((2 − c) mod 2);
11     h := h + 1;                      }
...    ...                              } as in Alg. 2, Line 5–10
16     i := i + ((d − 2) mod 2);        }
17     J := J + ((d − 1) mod 2);
18     I := I + ((d − 2) mod 2);
19     d := d xor_bitw (isOdd(ℓ − 1) xor a = 1);
```

Odd-sized Cells of Nano-programs

As depicted in Figure 5.4, all nano-programs for $2 \times 2 ... 4 \times 4$ grids exist. However, for the non-square sub-grids $(2 \times 3, 2 \times 4,$, and $3 \times 4)$ we need two versions: those beginning and ending at the longer side and those beginning and ending at the shorter side. For the 2×4 nano-programs, both versions exist: in addition to the nano-program 1232123_4

beginning and ending at the longer side, we can also define the 4×2 nano-program 1112333_4 (read from right to left cf. Fig. 5.4) beginning and ending at the shorter side. In contrast, a 2×3 sub-grid beginning and ending at the longer side would require a diagonal transition (being less local and requiring more than two bits).

Although odd-sized nano-programs are necessary if we want to support a global grid size with one or both side lengths being odd, we can completely avoid the non-existing nano-programs if we carefully plan where to place the $3 \times \{2, 3, 4\}$ nano-programs. Our key idea is, at most places, to allow only even-sized nano-programs. Every grid where both side lengths are even can be completely tessellated with only even-sized nano-programs. If one side length of the grid is even, and the other is odd, we can place all 3×2 and 3×4 nano-programs at that side of the grid opposite to the starting and ending point, as depicted in Figure 5.5. We have to control the traversal order of the grid (by specifying the initial direction $d = 3$ or $d = 2$) to make sure that the even-sized side length is opposite to the global starting and ending point (Figure 5.5, middle).

As depicted in Figure 5.5, right side, the situation is more tricky if both side lengths are odd. There, we have to place a column of 3×2 nano-programs at the right side and a single 3×3 nano-program at either of the ends of this column (in Figure 5.5, on the upper right corner). The Hilbert-sub curves at the upper and lower row are, unfortunately, not oriented such that the remaining 3×2 nano-programs can be positioned in a single row. The drawn layout with sub-curves oriented towards the middle is generated by bitwise logic operations.

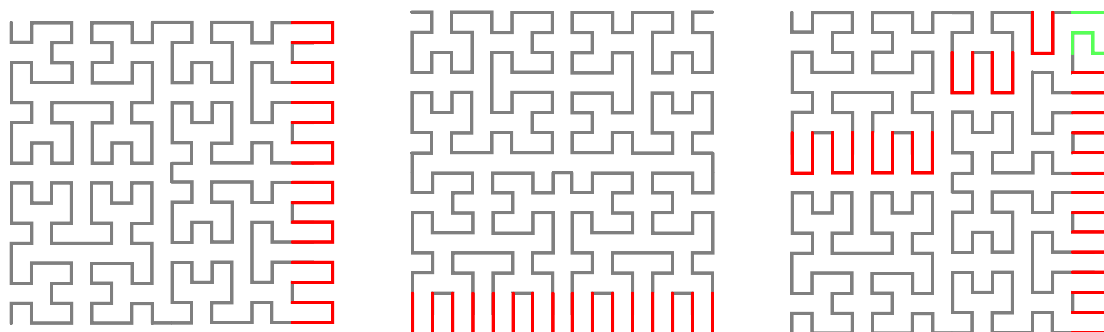


Figure 5.5: Placement of 3×2 (red) and 3×3 (green) Grids.

Severely Asymmetric Grids

If $\lfloor \log_2 n \rfloor < \lfloor \log_2 m \rfloor$ like in Figure 5.6 where $\lfloor \log_2 n \rfloor = 2$ and $\lfloor \log_2 m \rfloor = 3$ we put a number $m' = \lceil m/2^{\lfloor \log_2 n \rfloor} \rceil$ of independent curves side by side, where the first has width $m - \lfloor \log_2 n \rfloor \cdot (m' - 1)$ and the remaining have width $\lfloor \log_2 n \rfloor$. All the curves are anticlockwise (initial $d = 2$). With this setting it is guaranteed that the nano-programs of size $3 \times \{2, 3, 4\}$ can be placed exactly such that no diagonal transitions are needed. If $\lfloor \log_2 m \rfloor > \lfloor \log_2 n \rfloor$ we analogously put $n' = \lceil n/2^{\lfloor \log_2 m \rfloor} \rceil$ clockwise curves (initial $d = 3$) one above the other. Finally we note that starting with lower bounds for (i, j) different from $(0, 0)$ is also straightforward possible (cf. Figure 5.6 where we start at $(i, j) = (2, 0)$) at no extra cost per loop iteration. Degenerate loops where one or both variables iterate over one value only or no values at all (empty loops) are also considered, but these extensions are left out in our pseudo-code for clarity and space restrictions.

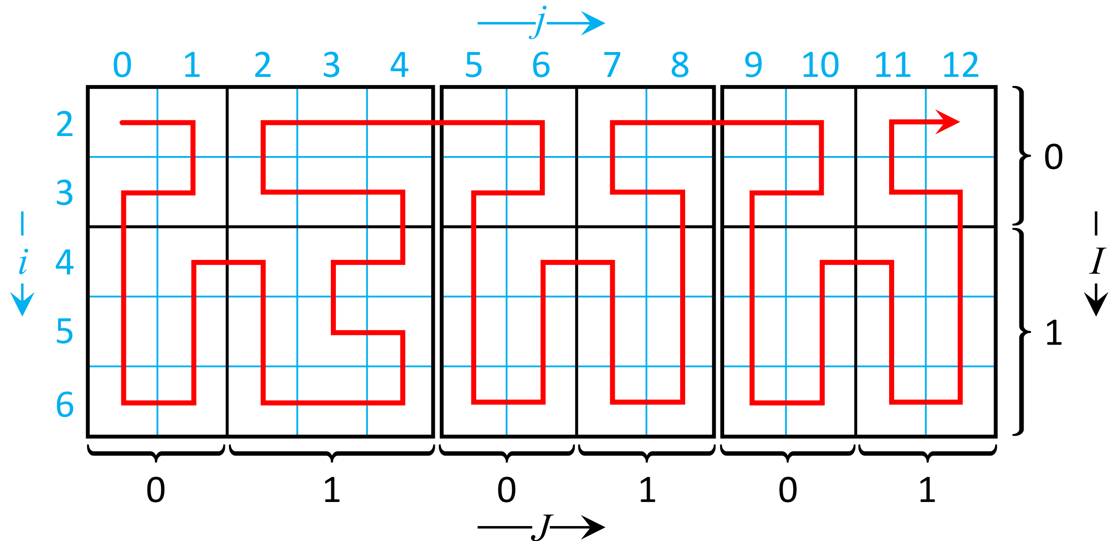


Figure 5.6: FurHilbertFor $(i, j) \in \{2, ..., 6\} \times \{0, ..., 12\}$.

5.6 Overall Architecture

Having made our code to generate loop iterations completely non-recursive and having removed all restrictions on loop boundaries for i and j, we will from now on note the cache-oblivious loops following the FUR-Hilbert curve in our pseudo-codes as follows:

FurHilbertFor $(i, j) \in \{imin, ..., imax-1\} \times \{jmin, ..., jmax-1\}$
  do *process object pair* (i, j);

This is very analogously possible in the source-code for C, C++, etc. where we define preprocessor macros:

#define FurHilbertFor $(i, j, \textit{imin, imax, jmin, jmax})$
#define FurHilbertEnd (i, j)

of which the first contains Alg. 3, Line 1–5, the part of the code before the placeholder *process object pair* (i, j), and the second contains Line 7–19, including all extensions described in Section 5.5. The C++-file can be downloaded from https://informatik.univie.ac.at/dm/downloads/.

The application of these preprocessor macros is almost as convenient as the application of standard loops. FUR-Hilbert loops can be nested with other loops and with each other. The implementor of the host algorithm can choose freely the name and type of the iterator variables (for all types allowing to apply the operator "+") and parenthesize the loop body as usual with "{" and "}". In this case editors automatically apply appropriate indentation. The following lines generate the FUR-Hilbert curve in Figure 5.6:

```
int i, j; FurHilbertFor (i, j, 2, 7, 0, 13) {
    printf("%d %d\n", i, j);
} FurHilbertEnd (i, j);
```

This architecture not only makes our host algorithm clear and well structured, but it also has significant performance benefits. The whole host algorithm can be implemented in a single method. It can apply local variables for the management of our loops and all information needed in the host algorithm inside and outside of the cache-oblivious loop. These local variables can be assigned to registers by the compiler or upon user request (e.g., by the keyword **register**). Various optimizations, including extraction of loop invariants and loop unrolling, can be made fully automatic by the compiler. These options are all unavailable in a recursive implementation.

5.7 Concluding Remarks

In this chapter, we have proposed the Hilbert order, a space-filling curve that has been implemented as a preprocessor macro. In the next two following chapters, we apply the Hilbert curve to several algorithms. In chapter 6, we apply the Hilbert curve to

algorithms from the linear algebra and in chapter 7 to the similarity join problem.

Chapter 6

Applications of Cache-oblivious Hilbert Curve

On top of FUR-Hilbert loops, we implemented a number of algorithms (matrix multiplication, K-means clustering, Cholesky decomposition, and Floyd-Warshall) and a number of further algorithms from data mining, linear algebra, database systems, and other fields. In these algorithms, we additionally parallelized the FUR-Hilbert loops with OpenMP, marking these parallelized loops with **FurHilbertFor***, and the innermost loops with SIMD-parallelism using AVX2 (Advanced Vector Extensions) marking those with the comment "// SIMD."

6.1 Algorihtms

Matrix Multiplication

Matrix multiplications are vastly used in software implementations and have numerous applications[17]. The basic algorithm to multiply $B \in \mathbb{R}^{n \times p}$ and $C \in \mathbb{R}^{p \times m}$[1] can be implemented using a FUR-Hilbert loop because it has no general data dependencies. If p is too large e.g. for a cache-size of 32K say $p \gg s := 1024$, to fit at least a few of the rows of B and C^{T} fit into the L1 cache, it is necessary to decompose the matrices horizontally into groups of s (divisible by 4 to ensure alignment with cache lines) columns before applying the FUR-Hilbert loop:

[1] stored as C^{T}, additionally each row aligned to cache lines

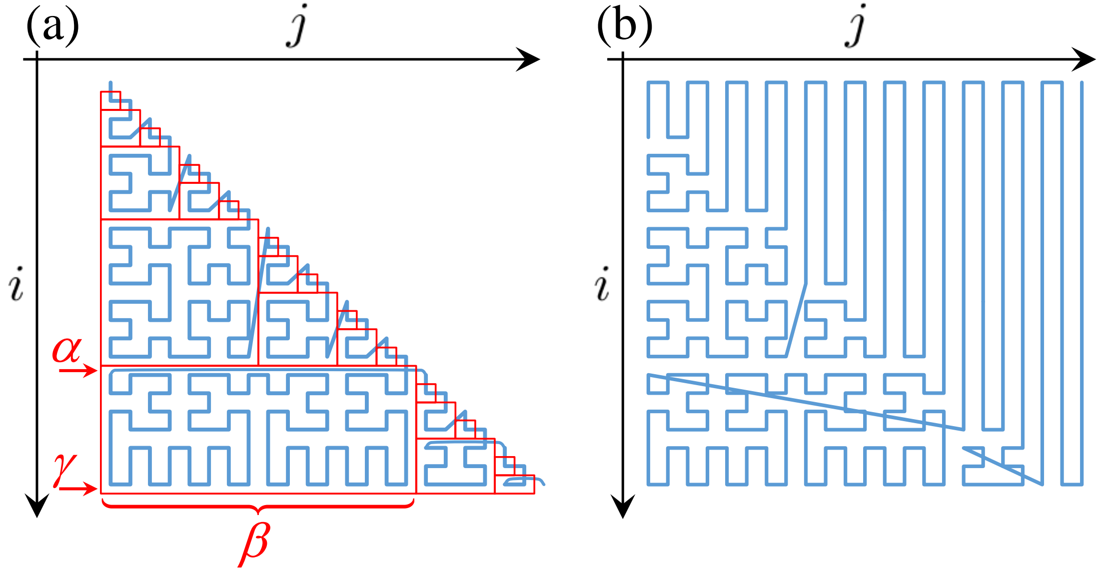


Figure 6.1: Traversal of Cholesky (a) and Floyd/Warshall (b)

for $K := 0$ **to** $p-1$ **stepsize** s **do**
  FurHilbertFor* $(i,j) \in \{0,...,n-1\} \times \{0,...,m-1\}$ **do**
    for $k := K$ **to** $\min\{K+s, p\} - 1$ **do**
      $a_{i,j} := a_{i,j} + b_{i,k} \cdot c^{\mathsf{T}}_{j,k}$;   } // SIMD

K-means Clustering

K-means, the most popular clustering algorithm, is like the related EM and K-medoid methods, implemented in a loop which convergences after alternately performing assignment and update steps. The expensive assignment step determines for each point x_i the distance to each cluster representative μ_j $(0 \le j < k)$ and assigns it to that cluster ID having minimal distance. We must keep track of the winner distance and the corresponding cluster ID for each point. This can be facilitated in a SIMD-parallel way by backpacking [25] the cluster ID in the least significant bits of the distance, noted $\langle dist, cID \rangle$:

FurHilbertFor* $(i,j) \in \{0,...,n-1\} \times \{0,...,k-1\}$ **do**
  double $h := ||x_i - \mu_j||$;
  $\langle dist_i, cID_i \rangle := \min\{\langle dist_i, cID_i \rangle, \langle h, j \rangle\}$;   } // SIMD

Cholesky Decomposition

Having numerous applications in data mining and simulation, Cholesky decomposition is an algorithm that factorizes a symmetric, positive definite matrix $A \in \mathbb{R}^{n \times n}$ into a left triangular matrix L such that $A = LL^{\mathsf{T}}$. It determines the entries $\ell_{i,j}$ as follows:

$$\ell_{i,j} := \frac{1}{\ell_{j,j}}\Big(a_{i,j} - \sum_{0 \le k < j} \ell_{i,k} \cdot \ell_{j,k}\Big) \text{ if } j < i; \;\; \ell_{i,i} := \sqrt{a_{i,i} - \sum_{0 \le k < i} \ell_{i,k}^2}$$

In the computation of element $\ell_{i,j}$ we read the whole matrix row $\ell_{j,*}$. This data dependency must be considered when changing the loop order of (i, j) either by space-filling curves or by parallelism. We achieve this by decomposing the lower left triangle matrix L into square blocks of a side length β being a power of two. The largest block of side length $\beta = 2^{\lfloor \log_2(n-1) \rfloor}$ is placed in the lower left corner of L starting from row $i_{\min} = \beta$ (cf. Figure 6.1(a); if the overall dimension n of the matrix is not a power of two, then the blocks at the bottom lines are suitably trimmed and become a non-square $\beta \times \gamma$ rectangle). Recursively, to each placed square of side length β we connect one with side length $\frac{\beta}{2}$ on the upper left corner and on the lower right corner until the triangle is completely tesselated when we reach at $\beta = 1$ or at some other defined basic resolution (for SIMD parallelism using AVX 1 or 2 we use 4×4 squares as basic elements, which degenerate to triangles at the diagonal of the matrix). Inside each block it is guaranteed that the data dependency plays no role since always $i_{\min} > j_{\max}$. We can apply FUR-Hilbert loops as well as SIMD and MIMD parallelism inside such a block but have to make sure that the blocks are ordered sequentially top-down and within the same row from left to right. The recursive block-generation is made iterative in the following pseudo-code which also considers the case of a matrix dimension n being not equal to a power of two:

$\ell_{0,0} := \sqrt{a_{0,0}}$;
for $\alpha := 1$ **to** $n - 1$ **do**
  $\beta := \alpha$ **and**$_{\text{bitw}}$ $-\alpha$; $\gamma := \min\{\alpha + \beta, n\} - 1$;
  FurHilbertFor* $(i, j) \in \{\alpha, ..., \gamma\} \times \{\alpha - \beta, ..., \alpha - 1\}$ **do**
    $\ell_{i,j} := \frac{1}{\ell_{j,j}}\left(a_{i,j} - \sum_{0 \le k < j} \ell_{i,k} \cdot \ell_{j,k}\right)$; // SIMD
  $\ell_{\alpha,\alpha} := \sqrt{a_{\alpha,\alpha} - \sum_{0 \le k < \alpha} \ell_{\alpha,k}^2}$;

The Algorithms by Floyd and Warshall

The Algorithm by Warshall [126] finds connected components (the transitive closure of the boolean adjacency matrix $A \in \mathbb{B}^{n \times n}$) in a directed or undirected graph. The standard algorithm uses three nested loops, resulting in an $O(n^3)$ algorithm:

$$
\begin{array}{l}
\textbf{for } i := 0 \textbf{ to } n-1 \textbf{ do} \\
\quad \textbf{for } j := 0 \textbf{ to } n-1 \textbf{ do if } a_{j,i} \textbf{ then} \\
\quad\quad \textbf{for } k := 0 \textbf{ to } n-1 \textbf{ do if } a_{i,k} \textbf{ then } a_{j,k} := \textbf{true};
\end{array}
$$

Although the data dependencies are actually similar as in our previous example, Cholesky decomposition, it is not possible to restrict the operations to one half of the matrix only. The upper and lower triangle can then be decomposed in larger blocks similar to Cholesky, which are subject to MIMD-parallelism and traversed in an order based on space-filling curves. The operation in the innermost loop can be transformed into a logical OR-operation, which can be executed by SIMD-parallel operations. This is particularly attractive since AVX allows us to perform up to 256 such operations simultaneously on a single core.

$$
\begin{array}{l}
\textbf{for } \alpha := 1 \textbf{ to } n-1 \textbf{ do} \\
\quad \beta := \alpha \textbf{ and}_{\text{bitw}} -\alpha; \quad \gamma := \min\{\alpha + \beta, n\} - 1; \\
\quad \textbf{FurHilbertFor}^* \ (i, j) \in \{\alpha, ..., \gamma\} \times \{\alpha - \beta, ..., \alpha - 1\} \textbf{ do} \\
\quad\quad \textbf{if } a_{j,i} \textbf{ then } \forall k\colon a_{j,k} := a_{j,k} \vee a_{i,k}; \ \text{// SIMD} \\
\quad \textbf{for}^* \ j := \alpha \textbf{ to } \gamma \textbf{ do} \\
\quad\quad \textbf{if } a_{j,\alpha} \textbf{ then } \forall k\colon a_{j,k} := a_{j,k} \vee a_{\alpha,k}; \ \text{// SIMD}
\end{array}
$$

The overall traversal scheme of the two outer loops (i and j) is depicted in Figure 6.1(b). The algorithm by Floyd operates on a weighted matrix $A \in \mathbb{R}^{n \times n}$ and uses the same algorithmic pattern like Warshall. The line:

$$\textbf{if } a_{j,i} \textbf{ then } \forall k\colon a_{j,k} := a_{j,k} \vee a_{i,k}; \ \text{// SIMD}$$

is replaced by:

$$\forall k\colon a_{j,k} := \min\{a_{j,k}, a_{j,i} + a_{i,k}\} \ \text{// SIMD.}$$

6.2 Experimental Evaluation

Experiments have been performed on Intel Xeon E5-2680v3 CPU (2.5GHz, 12 cores) with 256GB RAM and Debian GNU/Linux 8 (Jessie) as the operating system. Each core is associated with 64 KB of L1 and 256 KB of L2 cache. The last level cache (LLC) is the shared L3 cache with a size of 30 MB. All measurements are averaged over 20 runs using double precision arithmetics.

Matrix Multiplication

Our algorithm FUR-Hilbert, as discussed in Section 5.6, is implemented in C++ and compiled with GCC version 4.9.2. We compare our algorithm to the algorithm "TifaMMy" for matrix multiplication based on the Peano Curve introduced by Bader et al. [12, 61] (source code has been obtained by the authors compiled with ICC version 16.0.3). Furthermore, we compare our algorithm to the specifically for Intel processors hardware- and hand-optimized Intel MKL library (https://software.intel.com/en-us/intel-mkl) version 11.3 (operation: dgemm). Intel MKL can accelerate popular frameworks like Apache Spark, Pythons NumPy, and SciPy. As a baseline, we also compare to the classical matrix multiplication coded in c++ transposing the input matrix for improved cache locality. The code has been auto-vectorized using the GNU C++ compiler.

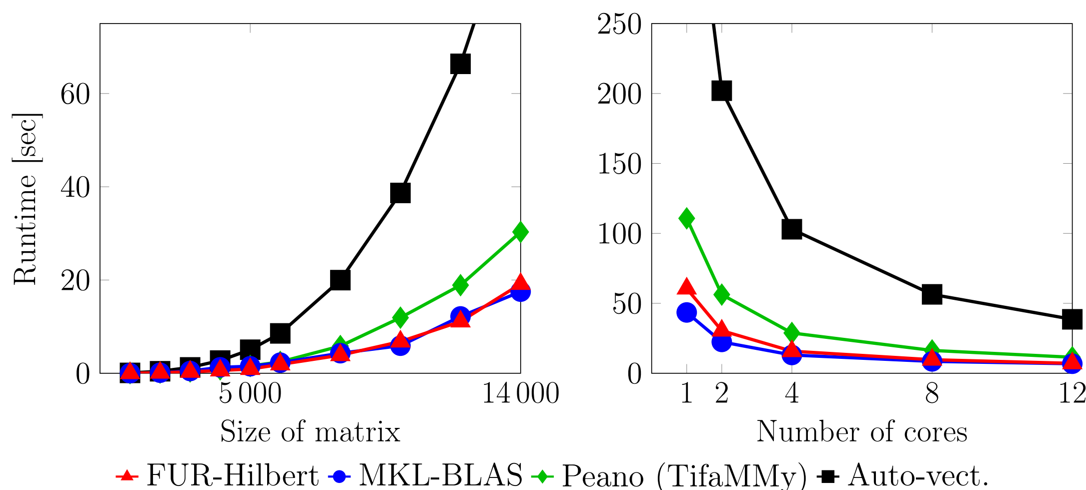


Figure 6.2: Performance of Matrix Multiplication (Xeon).

Figure 6.2 (left) displays the runtime in seconds varying the size of the input matrices from 1 000 to 14 000. For larger problem sizes, the highly optimized MKL library

slightly outperforms FUR-Hilbert. It processes the largest matrix in 17.50 seconds. Our algorithm needs 19.15 seconds and is at least 30% faster than the approach by Bader et al. (Peano). Peano requires 13.15 seconds. Our approach is more than ten times faster than auto-vectorization, which needs 1.7 minutes. Figure 6.2 (right) displays the runtime varying the number of threads. All techniques process two matrices with 10 000 elements each. All methods profit a lot from multi-threading. Auto-vectorization and Peano show almost linear speed-up, albeit at a low level of overall performance. FUR-Hilbert shows similar speed-up characteristics as MKL-BLAS.

Matrix Multiplication on a Manycore-System

The experiments have been performed on Intel Xeon Phi 7210 (KNL), with a processor base frequency of 1.3 GHz and 64 cores. We are using the memory mode "Cache Mode" and the cluster mode "All2All". Each core is associated with a 32 KB L1 data cache and shares 1 MB of L2 cache with another core on the same tile. In the "Cache Mode", the MCDRAM behaves as a memory-side direct-mapped cache in front of DDR4, which can be seen as a high bandwidth/high capacity L3 cache. Figure 6.3 (left) demonstrates the runtime spent for various problem sizes. Our algorithm outperforms MKL-BLAS for matrix sizes smaller than 9 000. For the largest problem size of 15 360 MKL-BLAS is around 14% faster and needs 5.87 seconds, whereas our approach needs 6.86 seconds. Nevertheless, our approach is 3.73 times faster than Peano ("TifaMMy"), which has a runtime of 24.96 seconds. Figure 6.3 (right) illustrates the runtime needed for different number of threads used. The matrix size is 7 680 and our approach takes only 0.918 seconds. Our approach is 76% faster than MKL-BLAS, which needs 1.62 seconds, whereas "TifaMMy" needs 3.47 seconds and the auto-vectorized approach needs 15.14 seconds.

Matrix Multiplication on Laptop

The experiments have been performed on MacBook Pro (13 inches, late 2016) with a 2.0 GHz dual-core Intel Core i5. Each core is associated with a 128 KB L1 cache, 512 KB L2 cache, and 4 MB shared L3 cache. Unfortunately, we could not run "TifaMMy" on macOS Sierra since it was not designed to run on multiple Mac OS X platforms. The tested implementations have been running independently from all other applications. We turned off background applications to get unaltered runtime measurements.

We are using 2 threads as the default configuration since we have only 2 cores. The 4 threads measurement is tested with the support of hyper-threading. We can see a similar

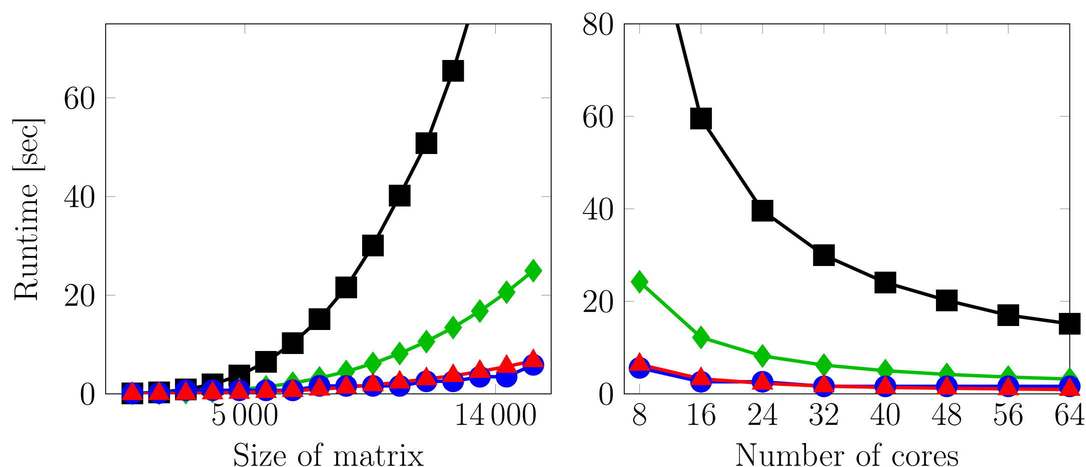


Figure 6.3: Performance of Matrix Multiplication on a Manycore System (Xeon Phi).

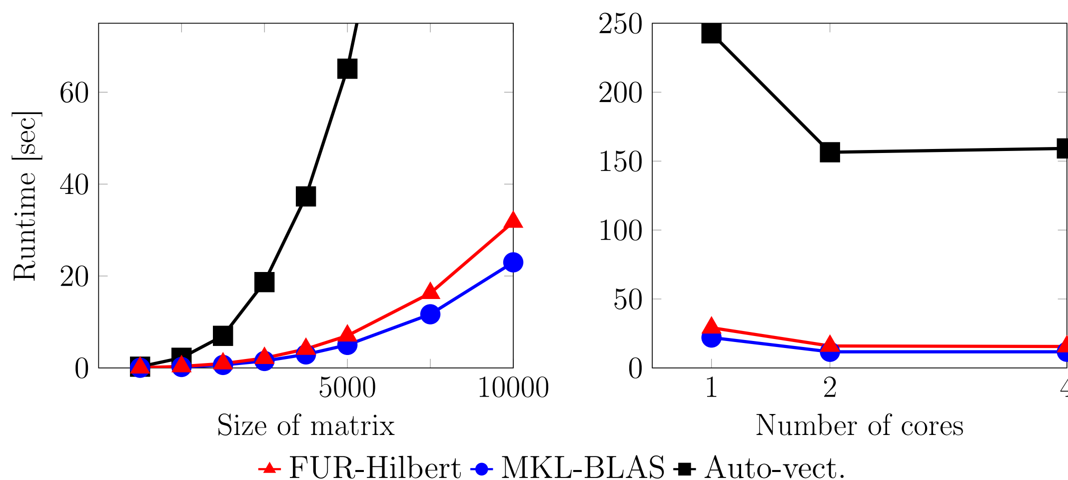


Figure 6.4: Experiments on matrix multiplication for a smaller cache size (Laptop).

behavior of the Hilbert curve on the laptop as it has on our servers. However, MKL-BLAS performs better on laptop systems. In Figure 6.4 left we tested various sizes of matrices up to a size of 10 000. For the multiplication of the largest matrices, the hand-optimized version of Intel MKL needs 22.97 seconds, whereas our Hilbert approach needs 31.76 seconds. In comparison to the auto-vectorized variant, our Hilbert implementation is roughly 9 times faster. For the threading experiment in Figure 6.4 (right) the default dimension of the input matrix is 8 000. Our version is close to the performance achieved by MKL-BLAS and even for hyper-threading 9 times faster than the auto-vectorized

approach.

Cache hierachy on Matrix Multiplication

The access time to memory is one of the bottlenecks for CPU core performance. Today's hierarchical cache structure reduces latency and hence speeds up the CPU clock. Here, we examine the cache hit footprint of our algorithm for the matrix multiplication. We are using Intel Vtune Amplifier XE 2017 to explore the cache access pattern of all algorithms among the L1, L2, and L3 hierarchy and calculate the cache hit rate for the L1 cache as: $\frac{L1_HIT}{L1_HIT+L1_MISS}$. Furthermore, we use "perf" version 3.16.7-ckt20 https://perf.wiki.kernel.org/ and the event *cache-misses:u* to detect a cache miss among the whole cache hierarchy. The `:u` tail excludes the kernel space and measures only the user space of the algorithms. The cache hit rate for "perf" is calculated as: $1 - \frac{cache\text{-}misses{:}u}{cache\text{-}references{:}u}$. We use the maximum number of threads (12) for the variation in problem size, and matrices of size 10 000 are processed for the threads' variation.

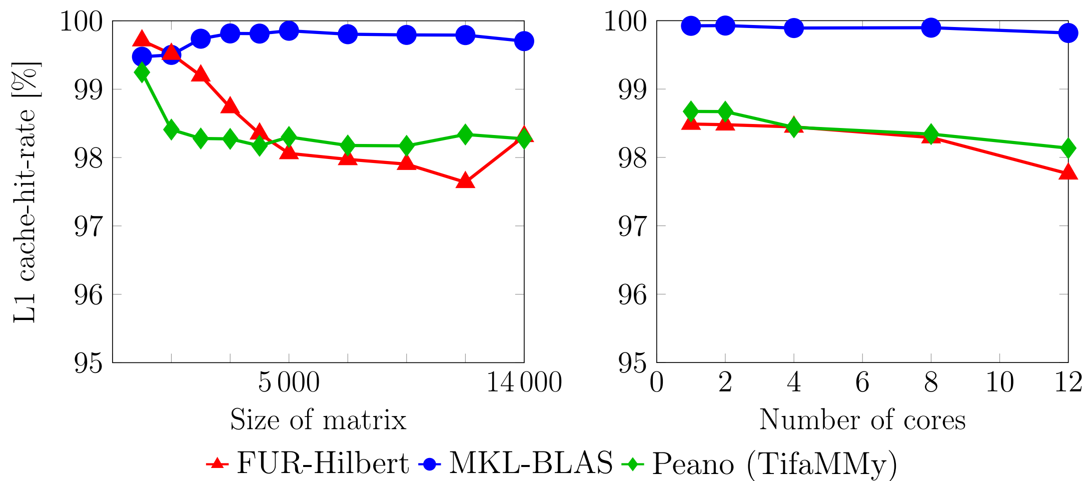


Figure 6.5: Cache-hit-rate of the matrix multiplication on L1 cache for different matrix sizes (left) and different thread sizes (right).

The Figures 6.5 to 6.8 illustrate the cache hit rate for each cache level respectively and the cache hit rate for the entire cache hierarchy. For the L1 cache-hit-rate (Figure 6.5), the hardware optimized MKL-BLAS is the most cache-efficient algorithm. Our algorithm performs well for small sizes and as well as Peano for larger sizes. All of the algorithms reached at least 97.5% of the L1 cache hit rate. For the L2 cache (Figure 6.6) hit rate, our algorithm performs best and remains within a lower bound of 94%

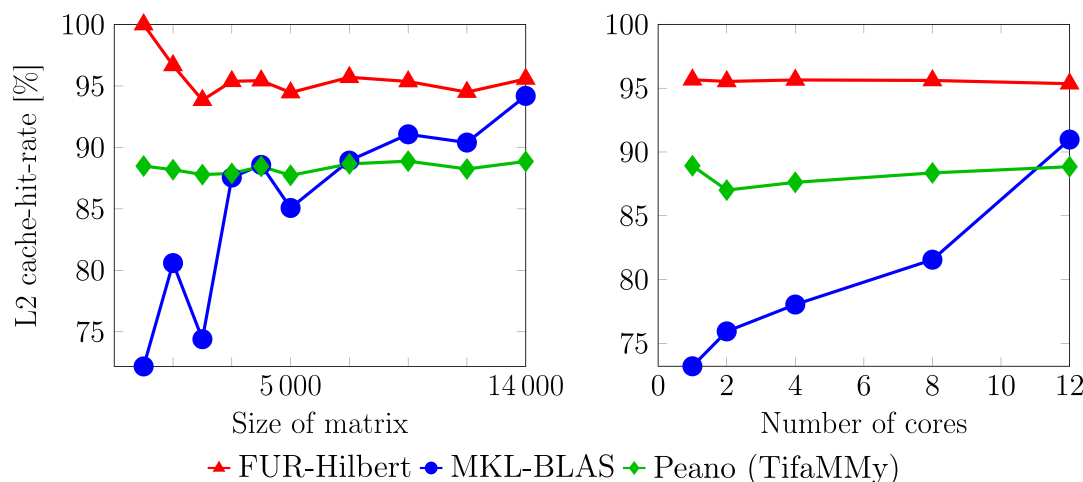


Figure 6.6: Cache-hit-rate of the matrix multiplication on L2 cache for different matrix sizes (left) and different thread sizes (right).

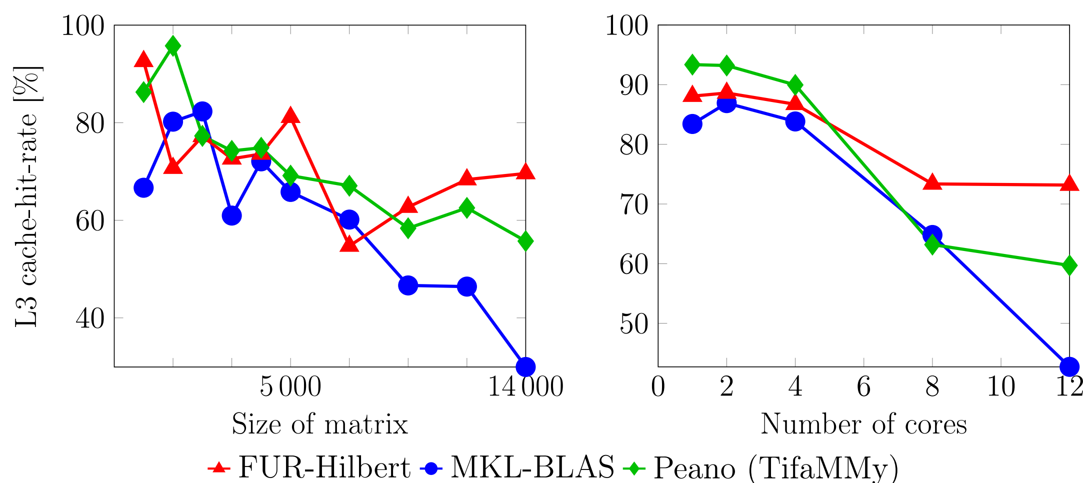


Figure 6.7: Cache-hit-rate of the matrix multiplication on L3 cache for different matrix sizes (left) and different thread sizes (right).

whereas Peano and MKL-BLAS have lower bounds of 87% and 72%. The last level cache (LLC), illustrated in Figure 6.7, is the slowest but largest in the cache hierarchy. Our algorithm shows good performance and uses most of the LLC for large matrix sizes and the maximum number of threads, whereas MKL-BLAS drops down in cache hit rates. The Peano algorithm performs relatively equal to our algorithm but shows a performance decrease for large problem sizes.

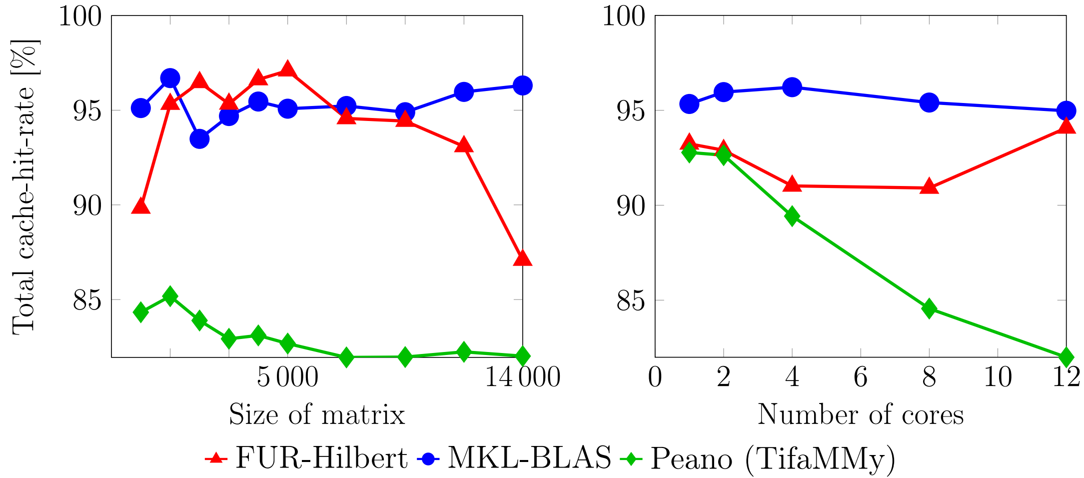


Figure 6.8: Cache-hit-rate of the matrix multiplication along the complete cache hierarchy for different matrix sizes (left) and different thread sizes (right).

We also captured the access pattern of memory access, which could be served by any of the cache levels in Figure 6.8. For the total cache hit rate, our algorithm competes with MKL-BLAS for sizes between 2 000 and 12 000 but drops down at both tails for matrix sizes of 1 000 and 14 000. Our algorithm clearly outperforms the Peano algorithm in both variations of matrix sizes and thread counts.

K-means

Here, we extended our K-means implementation [25] with the Hilbert curve. We use the same comparison methods as for matrix multiplication but exclude the Peano-curve based algorithm by Bader et al. [9, 12] since this approach is not designed to support K-means and is outperformed by MKL-BLAS on the task of matrix multiplication. The MKL library also does not include K-means. However, it can be used to speed up distance calculations. The MKL-based technique computes the scalar products between objects and cluster centers by matrix multiplication. As the runtime of K-means strongly depends on the number of iterations, we compare it for a fixed number of 5 iterations. As a primary setting, we consider a 20-dimensional data set with 1 048 576 data objects, and 24 000 clusters. A high number of clusters causes massive runtime but is practically relevant, e.g., for the coarse quantization step of the product quantization indexing technique [68].

Figure 6.9 (left) displays the runtime when varying the number of objects. Our technique processes 1 million objects in less than 8 seconds, while auto-vectorization

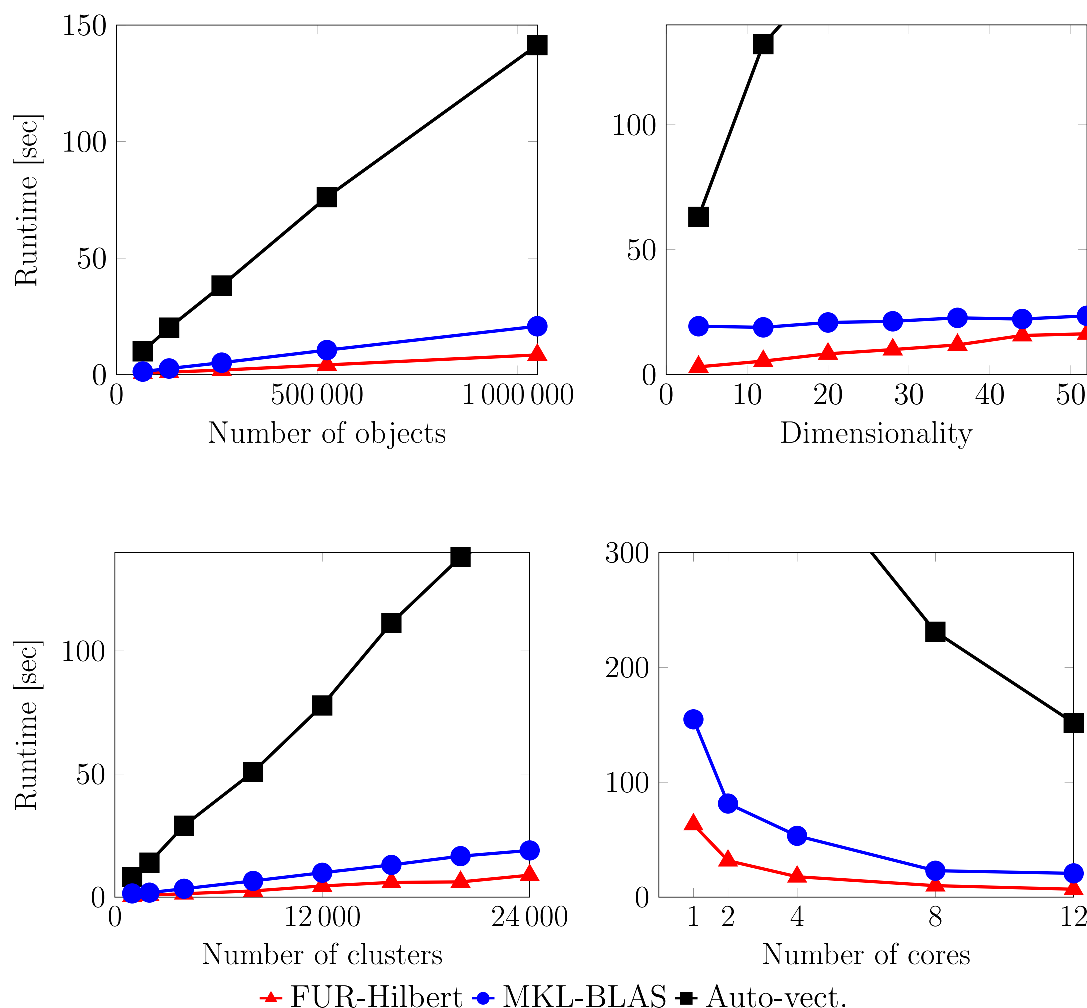


Figure 6.9: Experiments on K-means Clustering.

needs 2.88 minutes for 5 iterations of K-means. The next sub-figure varies the number of dimensions. The rightmost sub-figures display the speed-up, which is similar to matrix-multiplication.

We have also tested the hardware-optimized library DAAL (Intel Data Analytics Acceleration Library, https://software.intel.com/en-us/intel-daal), but DAAL does not compete in this setting. DAAL is a library of optimized algorithmic building blocks for data analysis and solves problems that are associated with “Big Data”. This includes regression, classification, or related problems, as well as clustering problems like K-means. We have been using DAAL with the current version of Intel Parallel Studio XE (version

2017 update 3). Unfortunately, DAAL cannot handle settings with a high number of clusters, where $K > 2\,050$. Even for $K = 1\,000$ or $K = 2\,000$ it performs worse than our auto-vectorization approach and needs 49.42 and 97.74 seconds for completion. We had been running DAAL in the batch processing mode, with the same settings used for our algorithms.

Cholesky Decomposition

All implementations of the Cholesky decomposition take a positive-definite matrix A and apply the decomposition of the form $A = LL^T$ in double precision. Our algorithm is implemented using C++ (compiled with GCC version 6.4.0 and OpenMP 4.5). The Linear Algebra PACKage (LAPACK) is a standard software library for numerical linear algebra. It provides routines for solving systems of linear equations and linear least squares. We compare our algorithm to the hardware optimized library LAPACK implemented in the Intel Math Kernel Library (MKL version 17.0 update 4, see https://software.intel.com/en-us/mkl/features/linear-algebra). We also compare our implementation to the well-known Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA version 2.8.0). The PLASMA project is a modern replacement of LAPACK with a software framework for developing asynchronous operations and out of order scheduling with the QUARK runtime. In contrast to LAPACK, which relies on BLAS level 2 calls, PLASMA uses BLAS level 3 calls, especially suitable for algorithms like Cholesky decomposition [30]. As a baseline, we also compare our algorithm to a classical implementation of the Cholesky decomposition, which has been automatically vectorized using the Gnu GCC compiler (Auto-vect.).

Figure 6.10 (left) displays the runtime in seconds on different dimensions of the input matrix varying from 1 000 to 12 000. For the largest matrix of 12 000, the hardware optimized MKL-LAPACK algorithm shows the best performance with 2.04 seconds, whereas our algorithm FUR-Hilbert needs 3.66 seconds. However, our algorithm is at least 20% faster than the PLASMA library, which needs 7.18 seconds and 17 times faster than the auto-vectorization taking 123.64 seconds to completion.

Figure 6.10 (right) displays the runtime varying the number of threads. Each algorithm processes a matrix of size 5 000. All algorithms profit from multithreading. Here again, for the highest number of threads, the MKL-LAPACK algorithm shows the best performance with 0.12 seconds. Our algorithm needs 0.39 seconds, and for this setting, we are six times faster than the PLASMA library, which needs 2.53 seconds, and ten

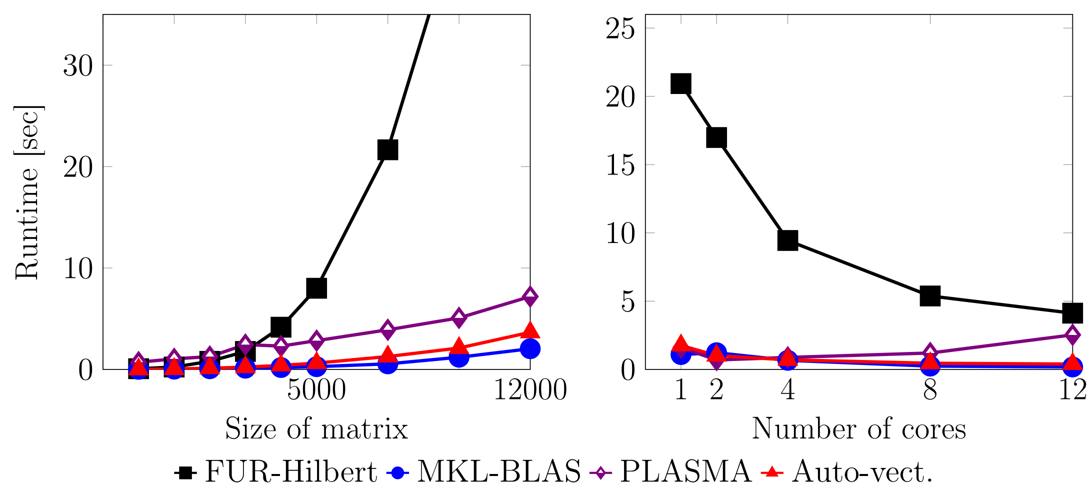


Figure 6.10: Experiments on Cholesky Decomposition.

times faster than auto-vectorization, which needs 4.12 seconds.

Algorithm by Floyd and Warshall

For the experiments on the algorithm of Warshall (described in Section 6.1) we have generated a graph with 3 densely connected subsets of nodes (clusters). Two nodes within a cluster are connected with a probability of 1% and two nodes in separate clusters are not connected, so we have 3 connected components. In Figure 6.11 we compare our packed FUR-Hilbert approach to the canonical approach. For a number of 40 000 nodes our approach is 1.81 times faster than the canonical implementation, where our algorithms needs 6.18 seconds and the canonical implementation 11.19 seconds. The same speedup of 1.81 remains for the variation in the number of cores used, where our algorithm spends 5.33 seconds and the canonical implementation 9.62 seconds.

Energy Efficiency

Energy is an indispensable resource with limited availability in all kinds of computing systems and critical in case of cost and availability. We measure the energy consumption of our Intel Xeon E5 with the power meter "Power HiTester 3334", which supports power integration measurements. This way of measuring is probably the most accurate approach to measure power and energy efficiency. There are also other APIs that allow us to read internal performance counters, such as Running Average Power Limit (RAPL) [127]. However, these were not available for our current hardware. Our power meter, the Hioki

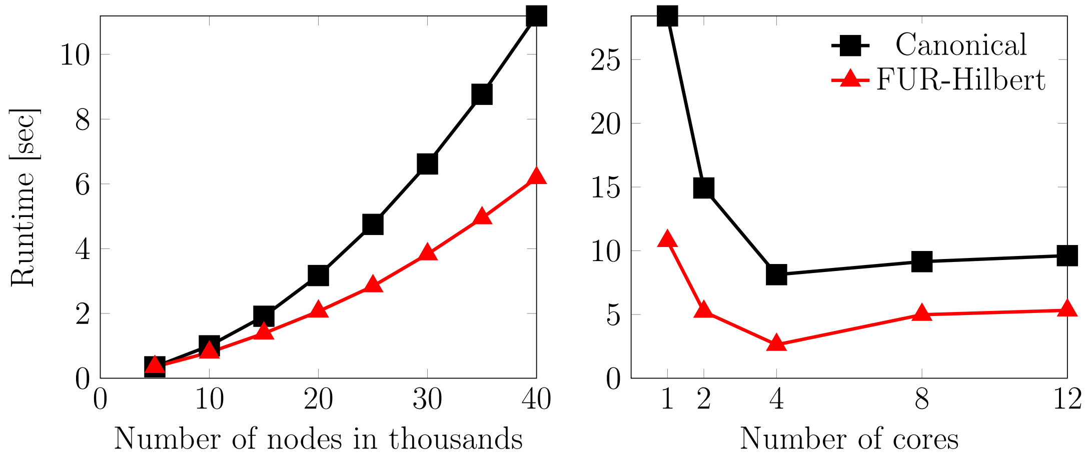


Figure 6.11: Experiments on the Algorithm by Warshall.

3334 AC/DC Power HiTESTER, has a measurement accuracy of ±0.1%. The 3334 is an AC/DC power meter that measures inrush current and power consumption, ideal for DC, and current and power integration applications to meet energy efficiency standards. We are using the serial port (RS-232) and a custom C API to communicate with the power meter, which allows us to measure the algorithms power consumption without any overhead. The code of our API is available to the public[2].

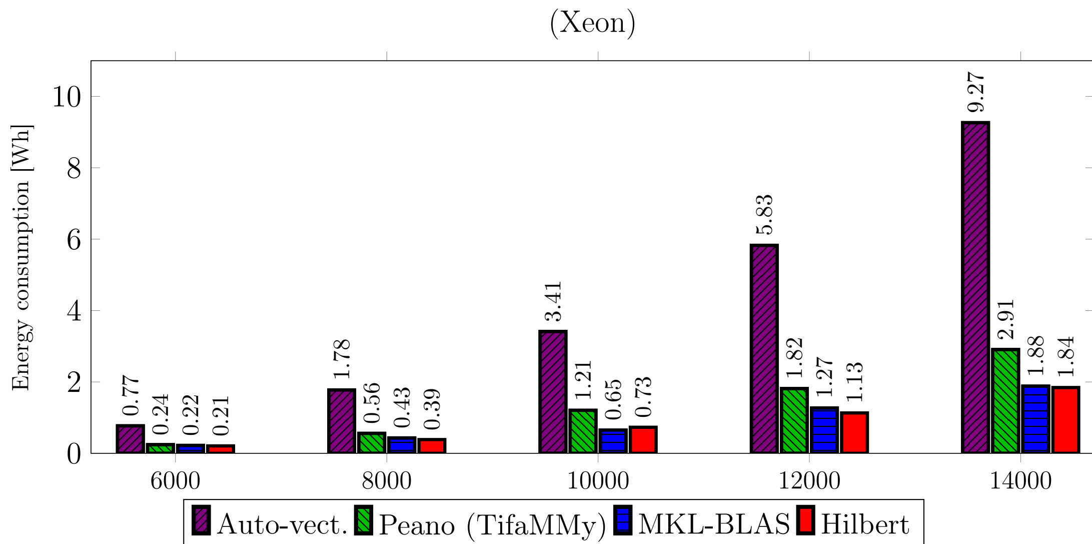


Figure 6.12: Energy efficiency for matrix multiplication.

[2] https://gitlab.cs.univie.ac.at/coloops/HIOKI-3334

Our algorithm is the most energy-efficient one for matrix multiplication, as illustrated in Figure 6.12. Our algorithm slightly outperforms the other algorithms for varying matrix sizes with one exception, where the problem size is 10 000. For other problem sizes, the gain over MKL-BLAS is between 2% and 11% and over Peano ("TifaMMy") between 16% and a factor of 160%. We are up to 5 times more energy-efficient than auto-vectorization, but we left it out in Figure 6.12 for clarity of presentation.

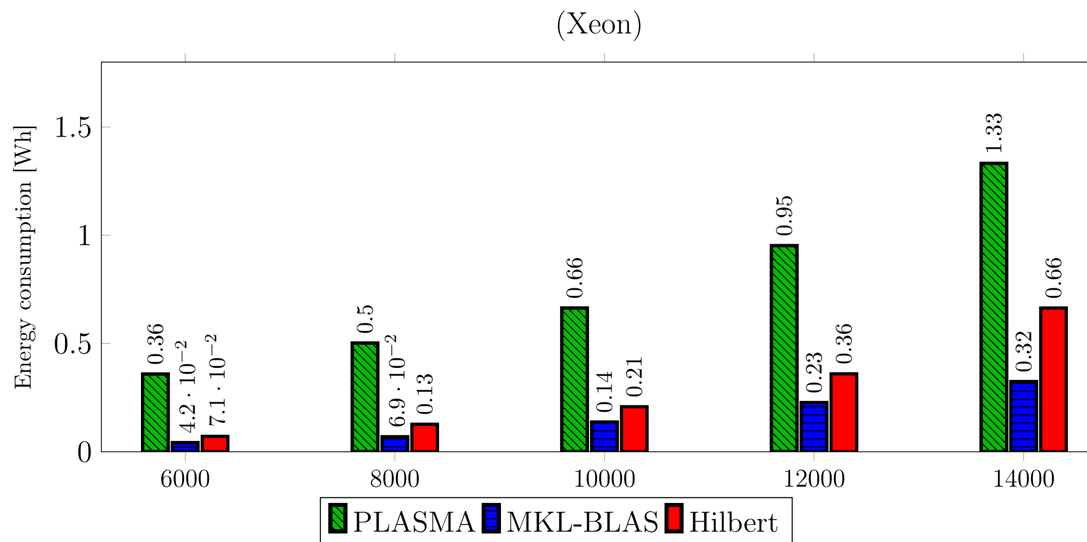


Figure 6.13: Energy efficiency for Cholesky decomposition.

Figure 6.13 displays the energy efficiency of the Cholesky decomposition. Here, we compare our algorithm with current state-of-the-art approaches on this microarchitecture. The Intel Math Kernel Library (MKL) version of LAPACK, named MKL-LAPACK, is the most energy-efficient algorithm. Our algorithm is between 2 and 5 times more energy-efficient than PLASMA. We are approximately 7 times more energy-efficient than auto-vectorization.

Our K-means algorithm outperforms MKL-BLAS in runtime and this is also true for energy consumption. For the largest problem size of 1 048 576, our algorithm consumes 0.726 Wh, whereas our BLAS implementation uses 1.923 Wh. Furthermore, the energy consumption of our auto-vectorization takes 13.882 Wh. BLAS consumes a factor of 2.64 more, and the auto-vectorized algorithm consumes even a factor of 19.12 more than our algorithm.

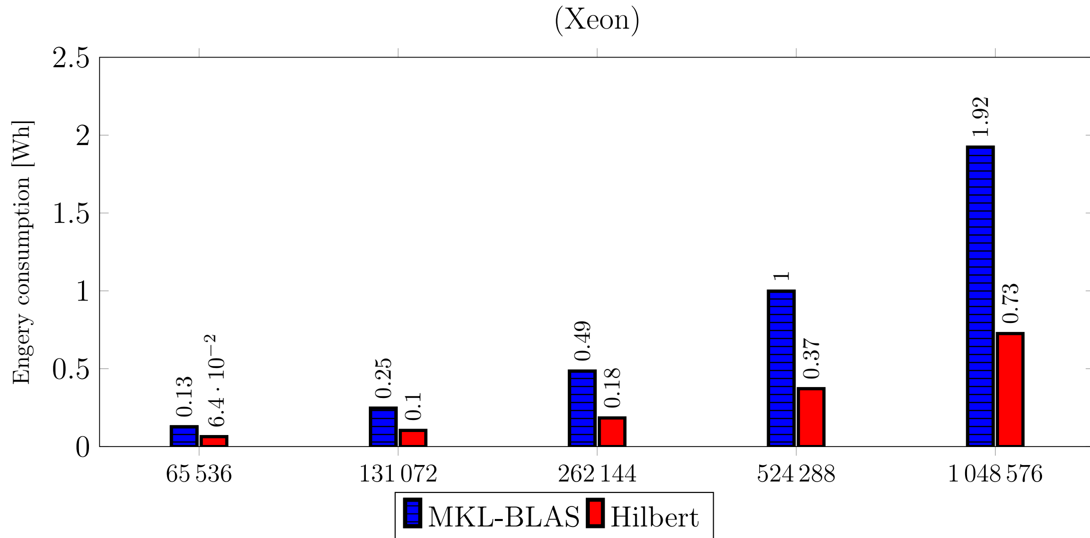


Figure 6.14: Energy efficiency for K-means clustering.

6.3 Related Work and Discussion

Cache-oblivious Algorithms

The notion of *cache-obliviousness* has been first introduced by Frigo et al. in 1999 [55]. A cache oblivious algorithm performs well on any type of multi-level memory hierarchy without knowing the structure and the hierarchy parameters, e.g., cache and memory size, transfer block size, and bandwidth.

Cache-oblivious algorithms [55] have attracted considerable attention as they are portable to almost all environments and architectures. Algorithms and data structures for basic tasks like sorting, searching, matrix multiplication, and specialized tasks like ray tracing [88] or homology search in bioinformatics [51] have been proposed. The two fundamental design patterns of cache-oblivious algorithms are localized memory access and divide-and-conquer. Space-filling curves (SFC) integrate both ideas. A SFC defines a 1D ordering of the points of an n-dimensional space such that each point is visited once. Probably the most widely used SFC is the Z-order due to its simple recursive scheme. The Hilbert and the Peano SFC provide better locality properties at the expense of more computational overhead. To theoretically formalize cache-obliviousness, Figo et al. introduced the assumption of an ideal cache, which is characterized by ideal page replacement and full associativity. The authors prove that algorithms designed with this

idealized setting in mind only degrade in performance by a constant factor in realistic settings such as LRU replacement strategy and set-associative caches. Most related to our work, Bader et al. proposed to use the Peano curve for matrix multiplication and LU-decomposition [9, 12]. Their algorithms process the input matrices in a block-wise and recursive fashion where the Peano curve guides the processing order and the memory access pattern. We considerably improve memory locality and runtime by introducing the Fast Un-restricted Hilbert curve. Classical SFCs are restricted to traverse data of a specific size, e.g. multiples of powers of two or three. Bader et al. use the Peano curve with zero paddings.

We introduce a Hilbert curve supporting arbitrary problem sizes, which is of general interest also for other applications. We propose a highly efficient iterative algorithm to compute the Hilbert values on the fly. Most existing non-recursive approaches are based on lookup tables, causing memory overhead, e.g., [36]. For the determination of a single Hilbert value, most iterative techniques are linear in the resolution, which corresponds to the number of iterations in the loop, thus causing substantial overhead. In contrast, our solution removes the recursion overhead of the Lindenmayer-System [54] and introduces compact nano-programs fitting into registers for the traversal of small patches. Our approach processes a single pair of indices i, j in constant time, iterate through the loop in linear time, and therefore causes negligible overhead. Recursive approaches, e.g., [27, 54] are associated with logarithmic worst-case time complexity for processing a single index pair and are not suitable to support comfortable loop programming.

Optimized Techniques for Specific Tasks or Hardware

In [11, 61] Bader et al. present variants of their algorithms for matrix operations. As in [9, 12], the general algorithmic scheme is recursive partitioning according to the Peano curve. To tailor the algorithm to specific hardware properties, small matrix blocks are processed with optimized assembler code. For every microarchitecture, comprehensive adjustments are required to reach performance competitive with optimized libraries like BLAS [47], LAPACK [6], DAAL or MKL. The library BLAS (Basic Linear Algebra Subprograms) provides basic linear algebra operations together with programming interfaces to C and Fortran. Specific implementations for various infrastructures are available, e.g., ACML for AMD Opteron processors or CUBLAS for NVIDIA GPUs. The Math Kernel Library (MKL) contains highly vectorized math processing routines for Intel processors.

In contrast to our work, these implementations are very hardware-specific and mostly

vendor-optimized. Moreover, they are designed to efficiently support specific linear algebra operations while aiming to support loop processing in general. Our experiments demonstrate that our cache-oblivious approach reaches a performance close to BLAS on the task of matrix multiplication and outperforms BLAS when applied to support K-means clustering.

As K-means probably is the most wide-spread clustering algorithm other highly optimized techniques for specific hardware have been proposed, e.g. for mobile devices [5], GPUs [15] or computing clusters [113]. Compared to such specialized techniques, it is out of the scope of this work as K-means clustering only serves as an exemplary host algorithm.

The algorithm by Warshall was also studied in a Bachelor thesis proposed by Naschenweng [90]. In this thesis, the student developed an implementation of a cache efficient version of the algorithm by Warshall using the Hilbert curve pattern to traverse a matrix in a triple-nested for-loop. The implementation was vectorized using AVX-512 SIMD intrinsics and experimentally evaluated on our Intel Xeon Phi. In the comparison against a reference implementation of the Boost library, his implementation showed superior performance in terms of runtime efficiency for certain blocking factors. In this thesis, the student also showed that data dependencies must be taken carefully into account and could lead to an incorrect result.

Energy Efficiency on Data Movement

The energy cost of data movement from memory to registers has been identified as one of the limiting factors for the development of efficient and sustainable exascale systems. In terms of energy, the cost is two orders of magnitude higher than the cost of computing a double-precision register-to-register floating-point operation [129].

In general, an L3 cache miss is approximately three times more expensive than an L2 cache miss, and an L2 cache miss is approximately three times more expensive than an L1 cache miss. In the Figures 6.5, 6.6, 6.7 and 6.8 we have examined the cache footprint of the matrix multiplication, where our algorithm avoids expensive cache misses most effectively. In cases of the runtime performance for the matrix multiplication, we observe that our algorithm is slightly behind MKL-BLAS (cf. Figure 6.2), but slightly ahead in cases of energy-efficiency (cf. Figure 6.12). We believe that this is due to our efficient cache access pattern induced by our Hilbert curve. The cache access pattern for the L2

cache in Figure 6.6 and the L3 cache in Figure 6.7 shows that our approach has a better cache hit rate for L2 and L3 caches.

6.4 Conclusion

We introduced Fast Unrestricted (FUR-) Hilbert in chapter 5, a new space-filling curve with the property that every arbitrary $n \times m$ rectangle can be filled by making only axis-parallel, single-step moves in a recursively bisected data space. The original Hilbert-curve has a similar property but is restricted to squares ($n = m$) where the side length n must be a power of two. Also, our algorithm that generates the FUR-Hilbert curve is highly efficient because it is non-recursive and has a constant worst-case complexity per generated pair of coordinates in contrast to $O(\log n)$ for previous methods. These two advantages make it particularly attractive to replace pairs of nested loops in important host algorithms (e.g., matrix multiplication and clustering) with our FUR-Hilbert curve, leading to cache-oblivious accesses of the corresponding objects. In extensive experiments, we demonstrated the superiority of such cache-oblivious loops.

Chapter 7

Cache-oblivious High-Performance Similarity Join

This chapter has a detailed look at shared memory databases with special attention to the similarity join. After a general introduction in section 7.1, we give a more specific introduction into the context of our work in section 7.2, where we formalize the similarity join including the problem setting and the filter- and refinement in more detail. In section 7.3, we will revisit the Epsilon Grid Order (EGO) and how we can apply the Hilbert curve in this algorithm. The full method FGF-Hilbert Join is explained in section 7.4. In our experiments in section 7.5, we evaluate the performance of the FGF-Hilbert Join, followed by a discussion including the related work in section 7.6. We give some concluding remarks in section 7.7.

7.1 Introduction

Finding pairs of similarities in data is one of the most crucial challenges in current database systems. These similarities are meaningful to many applications such as deduplicate detection, the ranking of search queries, data analytics, clustering, or data consolidation. In such use cases, the similarity is not exact. Instead, it is more an approximate similarity expressed for vector data as an ε-similarity. In such a similarity join, we are looking for pairs where the absolute difference between the two pairs is not higher than ε. A simple nested loop algorithm to test all possible pairs is quadratic in its runtime, which is very time consuming since the distance calculation for approximate pairs is expensive in terms of runtime performance. Considering Relational Database Management Sys-

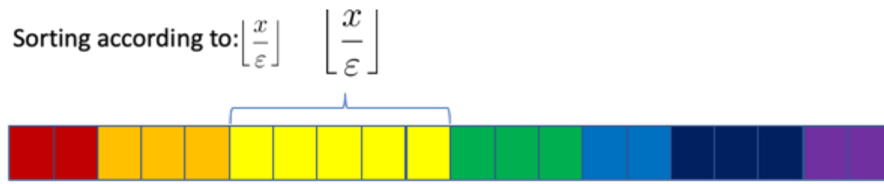


Figure 7.1: Sorting of 20 vector objects in the Epsilon-Grid Order (EGO).

tems (RDBMS) the solution should be generic and as performant as possible. It should be able to handle large and high dimensional datasets. A query in the computer science bibliography dblp (`https://dblp.uni-trier.de/`) for the term "similarity join" returns more than 400 results. These results contain solutions for different data types, such as streams, time-series, or images in different settings. For every data type, the similarity is expressed in a different similarity (or dissimilarity) measure. The edit similarity join [133] uses the edit distance, whereas the similarity of sets is calculated using the Hamming or Jaccard distance [44]. Every different setting has its own optimization principles and possibilities. This chapter focuses on vector data for millions of objects with up to 64 dimensions. The similarity join combines vectors based on the Euclidean distance. Typically, such algorithms apply a filter step and then refine the pairs of candidate vectors. In such a filter-step, the join candidates are excluded (e.g., by an index structure or sorting), which are not eligible for a join partner. In the following refinement step, the algorithm probe pairs of candidate vectors using nested loops.

An effective way to prune (or to filter) the space of vector data is to apply the ε-Grid Order (EGO), as shown in Figure 7.1. There, the data is sorted according to a grid consisting of ε steps. The value-range is expressed with the order of the colors in the rainbow. If two cells have the same color, then they are in the same ε-range. The 20 data points in the one dimensional dataset in Figure 7.1 are split into 7 grids cells. The EGO is an imaginary grid, which gets never materialized. The data points' ordering within one grid cell is determined from the *varepsilon*-grid in its next dimension, although the example in Figure 7.1 is restricted to only one dimension.

Based on the vector data from Figure 7.1 the possible join candidates can be visualized in an imaginary similarity matrix (c.f. Figure 7.2). In this example, a self-similarity join is considered. To avoid duplicates in the final result, only the upper triangle of the similarity matrix is considered. The discarded lower triangle is marked in grey in Figure 7.1. Data points that are more than ε distant can be safely discarded with the epsilon

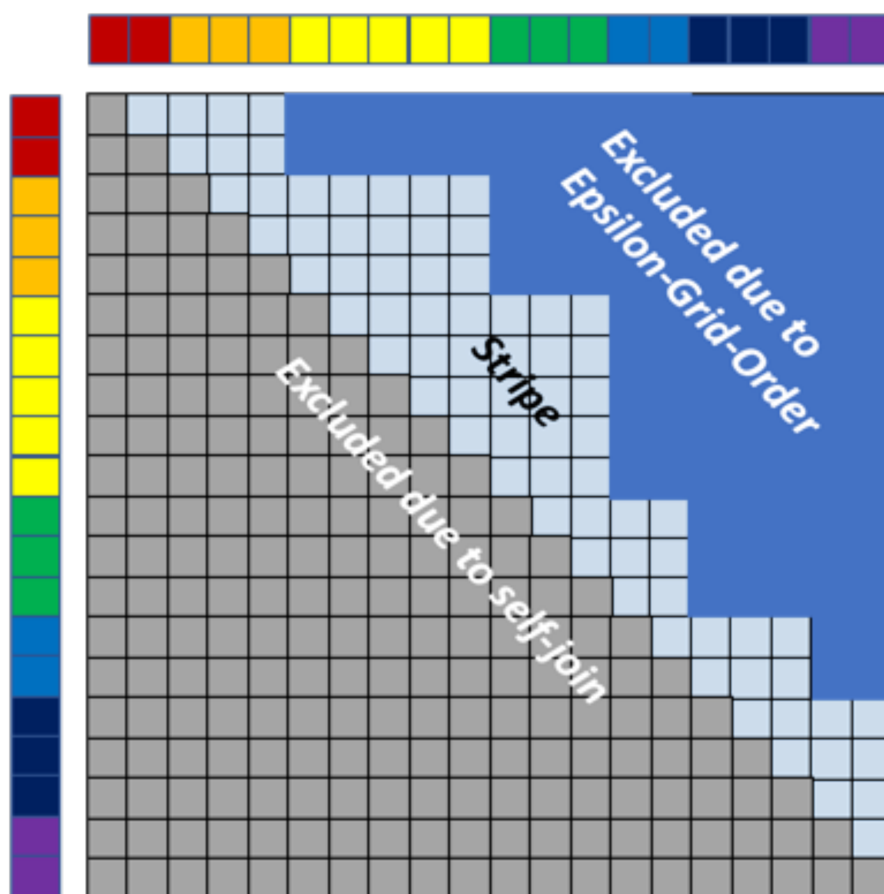


Figure 7.2: Imaginary similarity matrix for the ε-similarity join.

grid order. The region below the discarded points on the upper triangle is called stripe. The algorithm probe pairs of candidate vectors of the stripe region using nested loops in a refinement step. This chapter proposes to traverse these loops in a Hilbert order since it dramatically improves the data locality. As we demonstrate the easy transformation from conventional loops into cache-oblivious loops, we believe that many algorithms for complex joins and other database operations could be transformed systematically into cache-oblivious vectorized multi-core algorithms. In chapter 5, we introduced the Hilbert curve with the property that arbitrarily sized rectangles can be filled. Our novel space-filling curve Fast General Form (FGF) Hilbert combines the locality properties of FUR-Hilbert with a filter-and-refinement technique to efficiently perform a similarity join.

7.2 Similarity Join

Join algorithms with a complex join predicate like band-, similarity-, and k-nearest-neighbor-joins are essential for data mining algorithms like clustering, classification of massive data [18, 20]. Typically, they operate in a multi-step paradigm with a filter and a refinement step. The filter step constructs an index or sorts the data and applies a *simplified* and *more efficient* variant of the join predicate. The subsequent refinement step traverses two nested loops to apply the *actual* join predicate to all candidate pairs. Such a filter-and-refinement paradigm is correct and complete if the result of the filter step is a superset of the final join result or, in other words, if the join predicate of the filter step is a lower bound of the final join predicate. However, most existing solutions like [19, 44, 69, 70] are not designed for main memory databases running on multi-core systems with a complex memory hierarchy. We address this research gap with a parallel technique for the similarity join with optimal cache locality.

Problem Statement

We consider the similarity join of one or two sets $\mathbf{X} = \{\mathbf{x}_0, ..., \mathbf{x}_{n-1}\}$ and $\mathbf{Y} = \{\mathbf{y}_0, ..., \mathbf{y}_{m-1}\}$ of d-dimensional vectors which is defined using a threshold ϵ:

$$\begin{aligned} \mathit{SimJoin}_\epsilon(\mathbf{X}, \mathbf{Y}) &:= \big\{ (\mathbf{x}_i, \mathbf{y}_j) \subseteq \mathbf{X} \times \mathbf{Y} \;\big|\; ||\mathbf{x}_i - \mathbf{y}_j|| \leq \epsilon \big\} \\ \mathit{SimSelfJoin}_\epsilon(\mathbf{X}) &:= \big\{ (\mathbf{x}_i, \mathbf{x}_j) \subseteq \mathbf{X} \times \mathbf{X} \;\big|\; ||\mathbf{x}_i - \mathbf{x}_j|| \leq \epsilon \big\} \end{aligned}$$

where $||\mathbf{x}_i - \mathbf{x}_j||$ is the Euclidean distance. We denote the components of the vector $\mathbf{x}_i$ by

$$x_{i,k} \quad \text{where } i \in \{0, ..., n-1\} \text{ and } k \in \{0, ..., d-1\}.$$

Our objective is a time- and space-efficient algorithm on modern multi-core processors with SIMD and MIMD (Single/Multiple Instructions Multiple Data) parallelisms supported by a complex memory hierarchy. While we focus our subsequent discussion on $\mathit{SimSelfJoin}_\epsilon(\mathbf{X})$ our technique is also applicable to the general $\mathit{SimJoin}_\epsilon(\mathbf{X}, \mathbf{Y})$.

Filter and Refinement

Multi-step query processing with filter and refinement steps is often performed on index structures like R-trees [28] supporting the join predicate (Euclidean distance, in our case). However, the index construction is expensive. Therefore, a good alternative is to sort the

data set according to a function which supports the join predicate. A simple example for such an ordering (denoted $\leq_{\text{simple}}$) is to select one of the components of the vector, e.g. the first dimension $x_{i,0}$:

$$\mathbf{x}_i \leq_{\text{simple}} \mathbf{x}_j \qquad :\Leftrightarrow \qquad x_{i,0} \leq x_{j,0}.$$

We can use a simple predicate $|x_{i,0} - x_{j,0}| \leq \epsilon$ as filter step and compute the actual Euclidean distance (refinement step) only for those pairs $(\mathbf{x}_i, \mathbf{x}_j)$ which pass the filter step. If the filter predicate is a lower bound of the actual join predicate, it is guaranteed that the result set is *complete* (no false dismissals). Our ordering function $\leq_{\text{simple}}$ guarantees that the candidates to be refined form contiguous sequences in the sorted set (*locality*). The order $\leq_{\text{simple}}$ uses one dimension for the filter step only, resulting in a bad filter selectivity. In this chapter, we use the well-known Epsilon Grid Order (EGO):

$$\mathbf{x}_i \leq_{\text{EGO}} \mathbf{x}_j \qquad \text{(cf. Definition 2)},$$

which partitions the d-dimensional data space into a grid of squares of side length ϵ, and orders these grid cells by numbering them lexicographically (cf. Section 7.3). Using this strategy, *all* dimensions are used in the filter step (to guarantee good *filter selectivity*) while maintaining the lower-bounding property of the filter step (*completeness*) and the candidates for refinement are adjacent in the sorted set (*locality*). Due to locality, the potential join partners of a given point $\mathbf{x}_i$ form a contiguous interval $[lb(i), ..., ub(i)]$ in the sorted data set such that only those candidate pairs $(\mathbf{x}_i, \mathbf{x}_j)$ with $lb(i) \leq j \leq ub(i)$ need to be considered. Note that i and j always refer to positions in the sorted data set throughout this chapter. For query processing we use this algorithm:

sort $\mathbf{X}$ according to $\leq_{\text{EGO}}$;
determine $lb(i)$ and $ub(i)$ for each $i \in \{0, ..., n-1\}$;
for $i := 0$ **to** $n-1$ **do** // Filter Step
  for $j := lb(i)$ **to** $ub(i)$ **do**
    if $||\mathbf{x}_i - \mathbf{x}_j|| \leq \epsilon$ **then** // Refinement Step
      report $(\mathbf{x}_i, \mathbf{x}_j)$;

Cache-conscious Algorithms

To gain the maximum performance, we exploit modern computing hardware by vectorization (Single Instruction Multiple Data, SIMD) and multi-core parallelization (Multiple

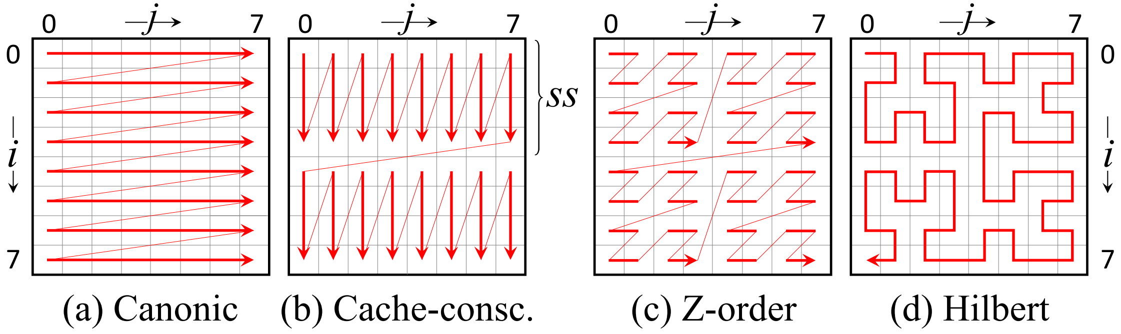


Figure 7.3: Strategies to Process Pairs (i, j) of Objects.

Instruction Multiple Data, MIMD), which is available today's inexpensive retail hardware.

However, the effect of both kinds of parallelism is severely limited by excessive accesses to main memory which is a centralized component in today's systems. Although alleviated through a number of cache memories (usually three levels, partially exclusive to single cores) and registers, these faster memories become only effective if the memory accesses are highly local. If our inner loop (j in our last code example) iterates over a number of vectors which do not fit into cache, this locality requirement is not fulfilled, and the cache does not well prevent us from the expensive memory accesses. An additional loop (I) makes our algorithm **cache-conscious**:

sort $\mathbf{X}$ according to $\leq_{\text{EGO}}$; determine $lb(i)$ and $ub(i)$;
for $I := 0$ **to** $n-1$ **stepsize** ss **do**
  for $j := lb(I)$ **to** $ub(I + ss - 1)$ **do**
    for $i := I$ **to** $I + ss - 1$ **do**
      if $||\mathbf{x}_i - \mathbf{x}_j|| \leq \epsilon$ **then** report $(\mathbf{x}_i, \mathbf{x}_j)$;

Provided that we have a single cache, and we have carefully chosen the step size ss such that ss+1 vectors of $\mathbf{X}$ fit into that cache, this *loop blocking strategy* is dramatically better: the number of transfers from main memory to cache is approximately reduced by the factor ss. The canonic order of two nested loops and the cache-conscious order can be compared in Figure 7.3 (a) and (b) where $ss = 4$.

Cache-oblivious Algorithms

Modern processors support a *hierarchy* of different memories with variants of the LRU strategy ("Least Recently Used"):

- L1, L2, and L3 cache (ordered by decreasing speed),
- a set of registers, even faster than the L1 cache,
- the translation look-aside buffer for the fast translation between virtual and physical memory addresses.

All these memories profit from the locality of accesses. When following the cache-conscious approach, a separate optimization for each of these memories is needed, resulting in a high number of loops. While we might be able to determine the pure hardware size of all these memories for a given hardware configuration, it is difficult to know (and subject to frequent changes) how much is available to our current join algorithm and not occupied by other processes.

To efficiently support the complete hierarchy of memories of (effectively) unknown sizes, we need a different concept: a **cache-oblivious algorithm** [55] is, unlike our above 3-loop construct, not optimized for a single, known cache size but supports a wide range of different cache sizes. The idea is to systematically interchange the increment of the variables i and j such that the locality of the accesses to both types of objects ($\mathbf{x}_i$ **and** $\mathbf{x}_j$) is guaranteed using space-filling curves like the Z-order (Fig. 7.3c) or Hilbert (d) curve [64]. In this work, we do not use space-filling curves as an index structure of the d-dimensional vector space (like e.g. [94]) but rather for the traversal order of objects in filter and refinement.

We propose the *FGF-Hilbert Join* (FGF for **Fast General Form**) applying our new variant of a cache-oblivious loop called FGF-Hilbert loop as follows:

sort $\mathbf{X}$ according to $\leq_{\text{EGO}}$; determine $lb(i)$ and $ub(i)$;
fgf hilbert loop $(i, j) \in \{0 \cdots n-1\} \times \{0 \cdots n-1\}$
  with constraint $lb(i) \leq j$ **and** $j \leq ub(i)$ **do**
    if $||\mathbf{x}_i - \mathbf{x}_j|| \leq \epsilon$ **then** report $(\mathbf{x}_i, \mathbf{x}_j)$;

7.3 Preliminaries

In this section, we introduce the most important and well-known building blocks of our technique, the Epsilon Grid Order, filter, and refinement techniques, followed by a discussion to reduce the overhead produced by the algorithm.

In our notation, we follow a view based on deterministic finite-state automata (DFA) and context-free grammars (CFG) to allow a rigorous mathematical treatment and prove the correctness of the FGF-Hilbert curve. Many iterative approaches to generate space-filling curves like [36] can be regarded as DFA, and many recursive approaches like [27] as CFG. See also section 5.3.

Epsilon Grid Order

The Epsilon Grid Order (EGO) is an ordering-based filter-refinement technique which has originally been proposed for hard-disk oriented database systems [19, 70]. The general idea is to lay a d-dimensional grid over the data space where the distance of the grid lines exactly corresponds to the radius ϵ of the join condition, by rounding-down: $\lfloor \frac{x_{i,k}}{\epsilon} \rfloor$. Thus it is guaranteed that join partners must be positioned in neighboring grid cells. EGO orders the grid cells canonically, i.e. the first attribute is the strongest ordering condition. If two vectors x_i and x_j are rounded to the same value in the first attribute, $\lfloor \frac{x_{i,0}}{\epsilon} \rfloor = \lfloor \frac{x_{j,0}}{\epsilon} \rfloor$, then the second attribute is considered, and so on.

Formally we define the ordering operator $\leq_{\text{EGO}}$ which is a total pre-order (a *reflexive, transitive,* and *total* relation; $\leq_{\text{EGO}}$ is not *anti-symmetric*):

Definition 2 (EGO)**.**

$\mathbf{x}_i \leq_{\text{EGO}} \mathbf{x}_j \quad :\Leftrightarrow \quad$ one of the following conditions holds:

(1) $\lfloor \frac{x_{i,k}}{\epsilon} \rfloor = \lfloor \frac{x_{j,k}}{\epsilon} \rfloor$ for all k with $0 \leq k < d$, **or**

(2) there exists a dimension D with $0 \leq D < d$ such that

$$\begin{array}{lcll} \lfloor \frac{x_{i,k}}{\epsilon} \rfloor & = & \lfloor \frac{x_{j,k}}{\epsilon} \rfloor & \text{for all } k < D \ \ \textbf{and} \\ \lfloor \frac{x_{i,D}}{\epsilon} \rfloor & < & \lfloor \frac{x_{j,D}}{\epsilon} \rfloor. & \end{array}$$

Iterative Generation of Hilbert Values

The simplest way of generating a loop enumerating the pairs (i, j) in Hilbert-order is to iterate over all possible Hilbert values h and to apply the inverse Hilbert function $(i, j) := \mathcal{H}^{-1}(h)$. The Hilbert value h is the order number of a pair (i, j) starting with

$h = 0$ at $(0, 0)$ and ending with $h = 63$ at $(7, 0)$, cf. Figure 7.3(d). Hilbert curves [64] need a square-like grid with a side length being a power of two. We round-up n to the next-higher power of two using $\lceil n \rceil_2 := 2^{\lceil \log_2 n \rceil}$.

```
for h := 0 to (⌈n⌉₂)² − 1 do
    (i, j) := H⁻¹(h);                          // cf. Appx. A
    if i < n and j < n then                    // (T₁)
        if lb(i) ≤ j and j ≤ ub(i) then        // (T₂)
            if ||x_i − x_j|| ≤ ε then          // (T₃)
                report (x_i, x_j);
```

(T_1) tests if i and j are generally in the allowed value range; (T_2) tests if i and j fulfill the constraints defined by the Epsilon Grid Order; and (T_3) performs the refinement step.

Many different methods [16, 36] have been proposed for the computation of the Hilbert function $h = \mathcal{H}(i, j)$ and its inverse $(i, j) = \mathcal{H}^{-1}(h)$. These approaches can be regarded as DFA producing output, the pair (i, j), bit by bit during state transitions ("*Mealy-DFA*"). The input (the Hilbert value h) is considered number in the 4-adic system which is processed digit by digit. The run-time is proportional to the number of digits, $O(\log n)$. This effort is due in each iteration of the loop. The DFA for $\mathcal{H}^{-1}$ is described in Section 5.3 and the recursive generation of Hilbert values in Section 5.3.

Discussion

Both iterative and recursive approaches share the disadvantage that they are inherently designed for square grids where the side-length is a power of two. If some general form, maybe a general rectangle (like [23]) or a more complex polygon is needed, the well-known solution is to produce Hilbert values for the smallest power-of-two square that contains the form and to ignore all values which are outside. The overhead of producing unnecessary pairs is high. Our filter step based on the Epsilon Grid Order produces a complex shape to be filled with the space-filling curve. With FGF-Hilbert loops, we propose a method that omits empty parts of arbitrarily complex shapes, cf. Section 7.4.

The recursive solution has the disadvantage that the actual refinement step of the join must be called from the two recursive functions $A(\ell)$ and $B(\ell)$. This causes a complex structure of the overall join algorithm and limits the compiler's optimization options (which is mostly focused on the optimization inside a single method). Like the iterative

```
1 algorithm FGF-HilbertJoin(ε)
2   sort (x_0, ...x_{n-1}) using ≤_EGO;
3   for i := 0 to n − 1 do
4     P_i := ε²/4 − 1/2 Σ_k x²_{i,k};
5     for each stripe s do determine lb^(s)(i) and ub^(s)(i);
6   for each stripe s do condense lb^(s) and ub^(s);
7   for each stripe s do
8     FGF-Hilbert loop (i, j) ∈ {0 ··· n−1} × {0 ··· n−1}
      with constraint lb^(s)(i) ≤ j and j ≤ ub^(s)(i)
      jumping over lb_ℓ^(s)(i) > j_max or j_min > ub_ℓ^(s)(i) do
9       if ⟨x_i, x_j⟩ + P_i + P_j ≥ 0 then report (x_i, x_j);
```

Figure 7.4: EGO-Join using a FGF-Hilbert loop.

solutions, the FGF-Hilbert loop is non-recursive.

7.4 The FGF-Hilbert Join

This section describes our solution for the similarity join, using the Epsilon Grid Order (EGO) as a filter step and our new FGF-Hilbert loop. We assume a modern multi-core CPU that allows MIMD-parallelism by multiple threads, and we used here Open-MP (easily adaptable to other architectures like CILK or POSIX-threads). For SIMD-parallelism, we rely on AVX-512, which allows using 32 vector-registers of 512 bit (i.e., vectors of 8 components in double-precision floating-point arithmetic). A straightforward adaptation to other AVX or SSE variants is also possible. We assume that our data is stored in the main memory $n \times d$ array (in row-major order), starting at a cache-aligned address.

We start by giving an overview of our algorithm along with its pseudo-code, cf. Figure 7.4. The building blocks are then discussed in detail in the subsequent sections. The algorithm starts by sorting the data set $\mathbf{X}$ according to $\leq_{\text{EGO}}$. Then we scan $\mathbf{X}$ to make some pre-computations: in Line 4, we compute the norms of the vectors which will be discussed in Section 7.4. In Line 5 and 6 we determine the lower and upper bounds $lb(i)$ and $ub(i)$ to plan which pairs (i, j) must be considered later in the refinement step, cf. Section 7.4. Finally, in Line 8, we traverse our cache-oblivious loop by a particularly fast variant of the Hilbert-curve, which is of constant time-complexity, non-recursive, and allows to omit areas of the space which are not covered by candidate

pairs (cf. Section 7.4). The refinement step in Line 9 is computed indirectly via the scalar product, cf. Section 7.4.

Determination of the Bounds

Previous works like [70] determine during the run of the recursive (divide-and-conquer) join algorithm if two subsets of EGO-ordered points can generate candidates. In contrast, our method plans carefully according to the epsilon grid order before the actual join which candidate pairs will be considered and stores the potential partners of a point $\mathbf{x}_i$ using lower and upper bounds $lb(i), ub(i)$. $ub(i)$ is the index of the last vector for which:

$$\mathbf{x}_{ub(i)} \leq_{\text{EGO}} \mathbf{x}_i + \begin{bmatrix} \epsilon \\ \vdots \\ \epsilon \end{bmatrix}.$$

The function $ub(i)$ is monotonically increasing. The monotonicity allows us to determine $ub(i)$ for all points in linear time, by an algorithm which follows after the proof:

Lemma 4 (Monotonicity of the function $ub(i)$)**.**

$$i \leq j \Leftrightarrow ub(i) \leq ub(j).$$

Proof. "$\Rightarrow$": let $i \leq j$. Then we know that $\mathbf{x}_i \leq_{\text{EGO}} \mathbf{x}_j$ in the EGO-sorted data set. According to Definition 2, we distinguish between two cases:

(1) $\lfloor \frac{x_{i,k}}{\epsilon} \rfloor = \lfloor \frac{x_{j,k}}{\epsilon} \rfloor$ for all $k < d$. In this case, we also know that $\lfloor \frac{x_{i,k}+\epsilon}{\epsilon} \rfloor = \lfloor \frac{x_{j,k}+\epsilon}{\epsilon} \rfloor$, because $\lfloor \frac{x_{i,k}+\epsilon}{\epsilon} \rfloor = \lfloor \frac{x_{i,k}}{\epsilon} \rfloor + 1$ and $\lfloor \frac{x_{j,k}+\epsilon}{\epsilon} \rfloor = \lfloor \frac{x_{j,k}}{\epsilon} \rfloor + 1$. Therefore, we have:

$$\mathbf{x}_i + \begin{bmatrix} \epsilon \\ \vdots \\ \epsilon \end{bmatrix} \leq_{\text{EGO}} \mathbf{x}_j + \begin{bmatrix} \epsilon \\ \vdots \\ \epsilon \end{bmatrix}, \text{ and thus } ub(i) \leq ub(j).$$

(2) In the second case of Definition 2, we know that there exists a dimension D such that:

$$\left\lfloor \frac{x_{i,k}}{\epsilon} \right\rfloor = \left\lfloor \frac{x_{j,k}}{\epsilon} \right\rfloor \text{ for all } k < D, \text{ and } \left\lfloor \frac{x_{i,D}}{\epsilon} \right\rfloor < \left\lfloor \frac{x_{j,D}}{\epsilon} \right\rfloor.$$

Likewise as in (1) we can conclude that

$$\left\lfloor \frac{x_{i,k}+\epsilon}{\epsilon} \right\rfloor = \left\lfloor \frac{x_{j,k}+\epsilon}{\epsilon} \right\rfloor \text{ and } \left\lfloor \frac{x_{i,D}+\epsilon}{\epsilon} \right\rfloor < \left\lfloor \frac{x_{j,D}+\epsilon}{\epsilon} \right\rfloor,$$

and thus that $ub(i) \le ub(j)$ holds also in this case.

The direction "$\Leftarrow$" of the equivalence is analogous. □

From the monotonicity of $ub(i)$ it follows that we can compute the complete set of upper bounds $ub(i)$ in a simple linear scan through the data set:

$ub(0) := 0$;
for $i := 0$ **to** $n-1$ **do**

while $\mathbf{x}_{ub(i)} \le_{\text{EGO}} \mathbf{x}_i + \begin{bmatrix} \epsilon \\ \vdots \\ \epsilon \end{bmatrix}$ **do** $ub(i) := ub(i) + 1$;

$ub(i+1) := ub(i)$;

This algorithm has two nested loops but is anyway $O(n)$ because each iteration of the inner loop starts with the last value $ub(i)$ of its previous iteration.

For a *general* join, we define analogously $lb(i)$ as first vector for which $\mathbf{x}_i - \begin{bmatrix} \epsilon \\ \vdots \\ \epsilon \end{bmatrix} \le_{\text{EGO}} \mathbf{x}_{lb(i)}$. The function $lb(i)$ is like $ub(i)$ monotonic. For a *self*-join, the lower boundary is the point itself ($lb(i) = i$), because we need to report the symmetric results $(\mathbf{x}_i, \mathbf{x}_j)$ and $(\mathbf{x}_j, \mathbf{x}_i)$ only once.

In Figure 7.5 we see an example, where in each row i the area between $j = lb(i)$ and $j = ub(i)$ is underlaid in gray (note that we are considering here a self-join, and in this special case always $lb(i) = i$). We call the gray area "the stripe". We will see in Section 7.4 that our FGF-Hilbert-curve partitions, like many other space-filling curves, the data space recursively into squares of a side length being a power of two (2^ℓ). We need to determine efficiently if such a bisection quadrant $\{i_{\min}, ..., i_{\max}\} \times \{j_{\min}, ..., j_{\max}\}$ can be completely excluded from the FGF-Hilbert loop generation (cf. Section 7.4). These boundaries are simply determined by rounding down and up to the next multiple of 2^ℓ:

$$i_{\min} = 2^\ell \cdot \lfloor i/2^\ell \rfloor, \quad i_{\max} = 2^\ell \cdot \lfloor i/2^\ell + 1 \rfloor - 1.$$

Therefore, we also determine condensed versions of $ub(i)$:

$$ub_\ell(i) := \max\{ub(i_{\min}), ..., ub(i_{\max})\},$$

as depicted on the right side of Figure 7.5. These values occupy only an additional space of $\lceil n \rceil_2$ integers and can be stored in the same (suitably enlarged) array as the original $ub(i)$, e.g. before them. Let us assume that the array UB stores the original $ub(i)$ at the position $\text{UB}[\lceil n \rceil_2 + i]$. The condensed values can also be produced in linear time:

$$\textbf{for } i := \lceil n \rceil_2 - 1 \textbf{ to } 1 \textbf{ do } \text{UB}[i] := \max(\text{UB}[2i], \text{UB}[2i+1]);$$

and then, $ub_\ell(i)$ is stored in $\text{UB}[\lfloor i/2^\ell \rfloor + \lceil n \rceil_2/2^\ell]$. By $\lceil n \rceil_2 := 2^{\lceil \log_2 n \rceil}$ we denote rounding up to the next higher power of two. In Figure 7.5, we can see on the right side the condensed versions of $ub(i)$, namely $ub_1(i), ..., ub_3(i)$. For instance, the entry $ub_2(0) = 9$ tells us that for the first $2^\ell = 4$ rows we need to consider only up to 9 columns and allows us to jump over the complete 4×4 quadrant in the upper right corner (green arrow from $(i, j) = (0, 12)$ to $(3, 12)$).

The interval $\{lb(i), ..., ub(i)\}$ can be split into sub-intervals $\{lb^{(0)}(i), ..., ub^{(0)}(i)\}$ to $\{lb^{(S-1)}(i), ..., ub^{(S-1)}(i)\}$ where our original $lb(i)$ and $ub(i)$ are now $lb^{(0)}(i)$ and $ub^{(S-1)}(i)$. The expected largest gap in the original interval is between $\mathbf{x}_i + [0, +\epsilon, ..., +\epsilon]^\mathsf{T}$ and $\mathbf{x}_i + [+\epsilon, -\epsilon, ..., -\epsilon]^\mathsf{T}$. The additional upper and lower bounds can be defined, determined, and condensed in the same way as $ub(i)$. Visually, this leads to the transition from one stripe in Figure 7.5 to a number S of stripes, each traversed by the space-filling curve (cf. Line 7 in Fig. 7.4).

Cache-oblivious Loops by FGF-Hilbert

We show how to traverse the marked gray area in Figure 7.5 by the red space-filling curve. We start by explaining how to generate the Hilbert values without any overhead and then elaborate on how to only process the relevant area as Line 9 in Figure 7.4). We derive this novel solution from the well-known Hilbert value generation by a Lindenmayer system. In contrast to the $O(\log n)$ recursive implementation, our solution is non-recursive and has a constant time complexity for each generated (i, j)-value pair.

Our fundamental idea is that all information that is stored in a stack of a recursive Lindenmayer implementation is already coded in the Hilbert value h and the current direction c. Consider for instance the situation depicted in Figure 5.3 where we have on

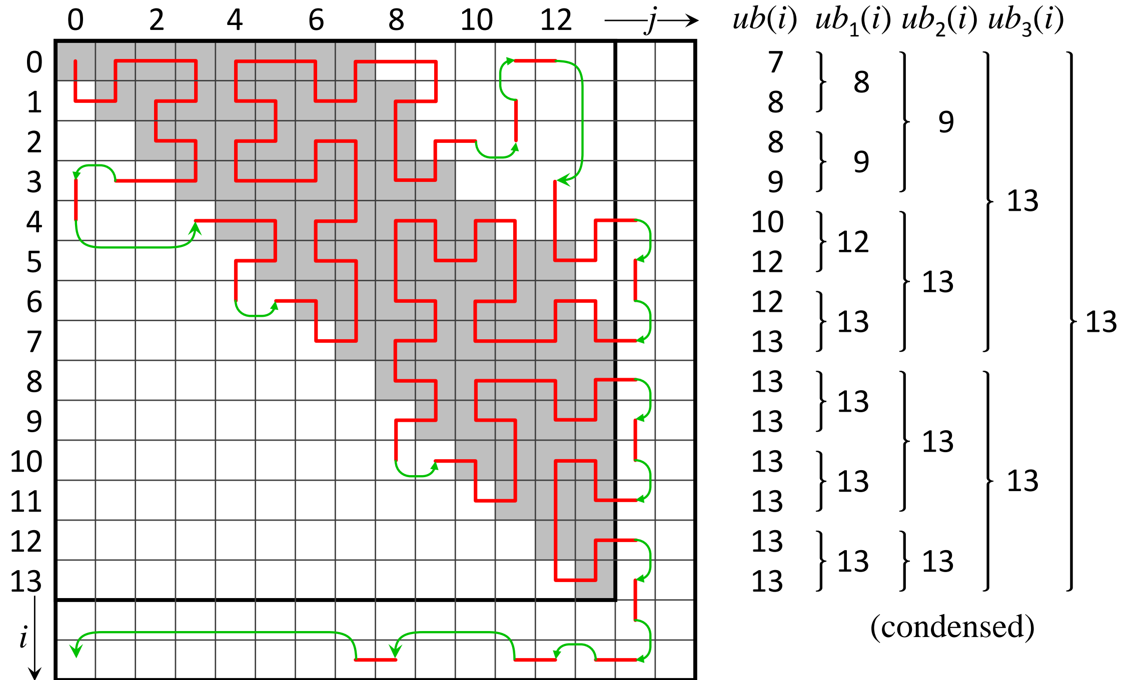


Figure 7.5: Planning Refinements: Upper bounds of intervals (right side) are stored and condensed; thus larger areas of the (i, j)-space can be efficiently discarded from loop traversal (left).

the stack labelA_2, labelA_3, and labelB_3 when generating pair $(i, j) = (7, 4)$. If we write the corresponding Hilbert value 47 in a four-adic system using the digits $\{0_4, 1_4, 2_4, 3_4\}$ we obtain $47_{10} = 233_4$ which exactly reflects the *numbers* k of the labels labelX_k on the stack. We prove by structural induction in Lemma 1 in Appendix B that the Hilbert value h, written as a 4-adic number, always equals the sequence of the *numbers* of the labels on the stack. Moreover, we can also recover the letters $X \in \{A, B\}$ of the labels to indicate if we are in function $A(\ell)$ or $B(\ell)$. This can be decided according to the combination of this number k and the $\mathit{parity}(c) \in \{\text{even}, \text{odd}\}$ of the current direction as proven in Lemma 2 in Appendix B.

With these two lemmata, we could simply make the functions $A(\ell)$ and $B(\ell)$ non-recursive by analyzing the simulated stack in the Hilbert value h digit by digit and putting the operations $\oplus, \ominus$, and $\triangleright$ in case distinctions. Our algorithm would benefit from this transformation by better optimization options. However, we would keep the $O(\log n)$ overhead of the recursive implementation because of the processing of h digit by digit (with a logarithmic number of digits).

To overcome this drawback, we focus our analysis on the forward steps in Figure 5.3: each production rule has three such $\triangleright$-steps, the first of them between the recursive calls at labelX$_0$ and labelX$_1$. The $\triangleright$-steps are preceded or followed by $\oplus$ or $\ominus$-operations. We call the combination of $\triangleright$ and the preceding or following $\oplus$ or $\ominus$ an *action* and number them by $a \in \{1, 2, 3\}$. There is no action $a = 0$ or $a \geq 4$ because no $\triangleright$-step exists before labelX$_0$ or after labelX$_3$. Action a is always executed between the recursive calls at labelX$_{a-1}$ and labelX$_a$. In the *simulated stack*, the 4-adic representation of h, we determine the action a as follows:

$$\begin{aligned} h &:= h + 1; \\ \ell &:= \left\lfloor \tfrac{1}{2} \log_2(h \ \mathbf{and}_{\text{bitw}} -h) \right\rfloor; \quad && // \text{ number of trailing } 0_4 \\ a &:= (h/4^\ell) \ \mathbf{mod}\ 4; && // \text{ fetch last digit} \neq 0_4 \end{aligned}$$

To obtain the correct action code, we first increase h. If we are at the end of one or more production rules, increasing h changes one or more digits 3_4 at the end of h (the *simulated stack*) 0_4. (cf. Lemma 1). In this case, the first digit (from right to left) $\neq 0_4$ pins down the level ℓ and the action a to be performed. Since a 4-adic digit corresponds to a pair of two bits, we determine the number ℓ of 00_2-bit-pairs at the end in constant time by the CPU-instruction TZCNT (Trailing Zero Bit Count), which is available in the instruction set extension BMI1 (Bit Manipulation Instructions) on Haswell and later CPU architectures. The same can be achieved by applying the bitwise **and**-operation to h and its negative complement $-h$, which gives us the largest power of two dividing h with no rest. The binary logarithm can also be determined by casting the result of $\mathbf{and}_{\text{bitw}}$ to a double-precision floating-point number and extracting the exponent, which is also very efficient and precise for $1 \leq h \leq 2^{52} - 1$, the greatest natural number that can be represented by double-precision floating-point numbers (according to IEEE-754) at no loss of precision. The corresponding operations are also available for vectors in AVX-512. The action code is finally extracted from h using $a := (h/4^\ell)\ \mathbf{mod}\ 4$.

In section 5.4 we prove that all $\oplus$ and $\ominus$-operations before and after the $\triangleright$-step (including those $\oplus, \ominus$ which are encountered when entering/leaving a recursive incarnation at the begin/end of a production rule) are subsumed by bitwise logic operations (exploiting the convention of C++ that boolean values $\{\mathbf{true}, \mathbf{false}\}$ are coded by the int values $\{1, 0\}$ which can be multiplied and subject to bitwise logic operations):

$$c := c\ \mathbf{xor}_{\text{bitw}}\ (11_2 \cdot (a = 3\ \mathbf{xor}\ \text{isOdd}(\ell))); \quad // \text{ before } \triangleright$$
$$c := c\ \mathbf{xor}_{\text{bitw}}\ (a = 1\ \mathbf{xor}\ \text{isOdd}(\ell)); \quad // \text{ after } \triangleright$$

We implement the $\triangleright$-step (cf. Def. 1(c)) in a way avoiding **if then else** which is expensive ("pipeline breaking") on today's processors:

$$\begin{array}{ll} & // \quad c \in \{\ 0,\ 1,\ 2,\ 3\} \\ i := i + (c-2)\ \mathbf{mod}\ 2; & //\ i := i + \{\ 0, -1,\ 0, +1\} \\ j := j + (c-1)\ \mathbf{mod}\ 2; & //\ j := j + \{-1,\ 0, +1,\ 0\} \end{array}$$

Note that the operation **mod** 2 preserves the sign and evaluates for $c \in \{0, 1, 2, 3\}$ to the values in the comments. Thus, only one variable (either i or j) is truly modified by ± 1. This results in the pseudo-code in Figure 2.

Lemma 5 (Complexity of LindenmayerNonRecursive)**.** The worst-case time and space complexity of our algorithm is $O(\mathit{const.})$ per loop iteration.

Proof. The body of our while-loop in Lines 5–11 of Figure 2 operates on 6 variables (i, j, h, c, ℓ, a) in 24 elementary operations ($+$, **and**, **mod**, etc.). Thus, the worst-case time and space complexity is constant per loop iteration. □

Constraint Enforcement

Constraints ($lb(i) \leq j$ **and** $j \leq ub(i)$) can be enforced in the tests T_1, T_2, T_3 (cf. Sect. 5.3) but an additional time overhead of Hilbert value generation is caused for the (i, j)-pairs which do not satisfy the constraint, even for those between n and $\lceil n \rceil_2$. Conventional nested loops like those in Section 7.2 and 7.2 do not yield this overhead. Therefore, we jump over bisection quadrants of an arbitrary level whenever it is guaranteed that no (i, j)-pair of the quadrant satisfies the constraint, see the green arrows in Figure 7.5.

When h has a number ℓ of digits 0_4 at its end, the non-recursive Lindenmayer algorithm performs an *action* at level ℓ and enters a new incarnation of a production rule at level $\ell - 1$ corresponding to a bisection quadrant of size $2^\ell \times 2^\ell$.

According to Def 1(c), if $c \geq 2$ ("looking right or down"), then we enter the quadrant at its upper left corner $(i_{\min}, j_{\min}) = (i, j)$ and we can determine the lower right corner $(i_{\max}, j_{\max})$ by *adding* $2^\ell - 1$ to i and j, respectively. If $c \leq 1$ ("looking left or up"), then we enter the quadrant at its lower right corner and we compute the upper left

corner by *subtracting* $2^\ell - 1$. We have to translate our constraint formulas into jump-over conditions operating on these corner coordinates. The **constraint** $i \leq j$ is translated into: **jumping over** $i_{\min} > j_{\max}$. If the latter is **true**, then the constraint is guaranteed to be **false** for all (i, j) pairs with $i_{\min} \leq i \leq i_{\max}$ and $j_{\min} \leq j \leq j_{\max}$, and we can jump to the last (i, j)-pair of the quadrant by adding $4^\ell - 1$ to h. To i or j we have to add or subtract the side length $2^\ell - 1$, depending on c and the parity of ℓ: If ℓ is even we then simply perform an $(2^\ell - 1)$-fold forward step ($\triangleright$) in direction c by applying:

$$\left.\begin{array}{l} j := j + (2^\ell - 1) \cdot \big((c-1) \ \mathbf{mod}\ 2\big); \\ i := i + (2^\ell - 1) \cdot \big((c-2) \ \mathbf{mod}\ 2\big); \\ h := h + (4^\ell - 1); \end{array}\right\} (2^\ell - 1) \text{ -fold } \triangleright$$

If ℓ is odd, then the directions change before and after applying the $(2^\ell - 1)$-fold forward step: c is changed from 0 to 1 (and reverse) and from 2 to 3 (and reverse) which is expressed in a simple formula: $c\ \mathbf{xor}_{\text{bitw}}(\ell\ \mathbf{and}_{\text{bitw}} 1)$.

The effect of jumping over bisection quadrants (marked by green arrows) can be studied using a 14×14-grid in Figure 7.5 where we jump over 12 quadrants of size 2×2, three quadrants of size 4×4 and one of size 8×8 which are excluded because $j < i$ or $j > ub(i)$ (outside the gray marked area). This strategy saves us a total of 144 generated (i, j)-pairs. The generation of condensed $ub_\ell(i)$ values depicted on the right side is discussed in Section 7.4.

In all pseudo-codes we note the FGF-Hilbert loop with constraints and jump-over conditions as follows:

FGF-Hilbert loop $(i, j) \in \{0, \cdots, n\} \times \{0, \cdots, n\}$
with constraint $lb(i) \leq j$ **and** $j \leq ub(i)$
jumping over $lb_\ell(i_{\min}) > j_{\max}$ **or** $j_{\min} > ub_\ell(i_{\min})$ **do** ...

For C and C++ code we provide a preprocessor macro having the constraint and jump-over conditions as parameters in which the names of iterator variables like (i, j) can be arbitrarily selected. For the jump-over condition the macro automatically provides variables for the quadrant boundaries $i_{\min}$, etc. and the level ℓ. Therefore, the use of the FGF Hilbert loop is as easy as in our pseudo-codes. Moreover, the compiler has full flexibility for optimization because all applied algorithms are non-recursive and thus the complete join algorithm is in a single method.

Euclidean Distance by Scalar Product

The Euclidean distance of the refinement step can be computed more efficiently through the scalar product:

$$\text{Scalar Product:} \quad \langle \mathbf{x}_i, \mathbf{x}_j \rangle := \sum_{0 \le k < d} x_{i,k}\, x_{j,k}.$$

We apply the binomial theorem $(a - b)^2 = a^2 + b^2 - 2ab$ to our join condition which yields:

$$||\mathbf{x}_i - \mathbf{x}_j||^2 = \sum_{0 \le k < d} (x_{i,k} - x_{j,k})^2 \le \epsilon^2 \Leftrightarrow$$

$$\sum_{0 \le k < d} x_{i,k}^2 + \sum_{0 \le k < d} x_{j,k}^2 - 2 \sum_{0 \le k < d} x_{i,k}\, x_{j,k} - \epsilon^2 \le 0 \Leftrightarrow$$

$$\underbrace{\sum_{0 \le k < d} x_{i,k}\, x_{j,k}}_{\langle \mathbf{x}_i, \mathbf{x}_j \rangle} + \underbrace{\frac{\epsilon^2}{4} - \frac{1}{2} \sum_{0 \le k < d} x_{i,k}^2}_{P_i} + \underbrace{\frac{\epsilon^2}{4} - \frac{1}{2} \sum_{0 \le k < d} x_{j,k}^2}_{P_j} \ge 0$$

After sorting but before performing the actual join, we can pre-compute the values $P_i := \frac{\epsilon^2}{4} - \frac{1}{2} \sum_k x_{i,k}^2$. In the refinement step, we then have to perform for each Euclidean distance only d fused multiply/add operations (plus two further additions) rather than d subtractions *and* d fused multiply/add operations. On today's architectures this pays off for $d \ge 3$ and is twice as fast in the limit $d = \infty$.

SIMD Parallelism (Vectorization)

Modern vector architectures like SSE or AVX allow us to compute arithmetic and logical operations on vectors consisting of 2, 4, 8, or 16 components. For instance AVX-512 allows 8 double-precision operations per clock cycle and processor core, which can even be a fused multiply/add operation of the form $R_1 := R_2 + R_3 \cdot R_4$, where $R_1, ..., R_4$ are selected from 32 available vector registers (per core). The fused multiply/add operations are ideal for the computation of the scalar product. Of equally high importance is to minimize data transfers between main memory caches and registers (which require, for instance, 4 clock cycles for a transfer from the L1 cache into a register; all other transfers are even more expensive). To guarantee good re-use of information once transferred into registers, we compute quasi-simultaneously 8×8 scalar products and reserve 16 of the registers for the intermediate results of these scalar products. Therefore, we change

Lines 3, 8, and 9 of our pseudo-code given in Figure 7.4 such that they operate with a step-size 8 for i and j, respectively. The refinement step in Line 9 is then computed simultaneously for a complete square of size 8×8. In addition to the direct report of $(\mathbf{x}_i, \mathbf{x}_i), ..., (\mathbf{x}_{i+7}, \mathbf{x}_{i+7})$ in Line 9 we compute the refinement steps for the corresponding upper triangle of the 8×8-block at the diagonal (e.g. $\langle \mathbf{x}_i, \mathbf{x}_{i+1} \rangle + P_i + P_{i+1}$ etc.) and report these join results.

MIMD Parallelism with OpenMP

Concepts like OpenMP or CILK can manage parallel threads for different cores of the CPU. For Line 2 in Fig. 7.4, we implemented a MIMD-parallel version of Quicksort, which opens new threads (using OpenMP tasks) in the recursive calls. Although other algorithms like Mergesort allow a higher degree of parallelism (perhaps including the additional opportunity of SIMD-parallelism), it turned out that the sorting step uses only a small percentage of the overall response time (which has been included in all experiments, cf. Fig. 7.8). The sorting step is not the main focus here in this chapter.

The loop in Line 3 and condensing (Line 6) can be straightforwardly parallelized using the construct “omp parallel for” where the data set is statically partitioned into equally-sized chunks assigned to parallel threads.

Before performing the loop over the stripes s and the FGF-Hilbert loops in Line 8 and 9 we partition the data sets into different chunks for the i-variable. As the intervals defined by $\{lb^{(s)}(i), ..., ub^{(s)}(i)\}, 0 \leq s < S$ are of different size for each i, we determine these chunks ch such that they yield approximately the same workload:

$$workload(ch) = \sum_{i \in ch} \Big(\sum_{0 \leq s < S} ub^{(s)}(i) - lb^{(s)}(i) \Big).$$

These load-balanced chunks are assigned to different threads using “omp parallel for”. Each thread performs S individual FGF-Hilbert loops with additional constraints and jump-overs enforcing i to be in the chunk assigned to the thread.

7.5 Experimental Evaluation

Experimental Setup

We share our code, and most of our test data to make our experiments transparent and comprehensible. Code and experimental data is available[1].

Hardware

Most experiments have been performed on Intel© Xeon Phi™ 7210 codename Knights Landing (KNL) with 1.3 GHz and 64 cores, 96 GB main memory, and CentOS 7.4.1708 as the operating system. A KNL processor socket has 32 active tiles, where each tile consists of two cores. Each core has 32 kB instruction and 32 kB data cache for L1 and supports AVX-512 SIMD instructions. The L2 cache of 1 MB is shared among two cores within a tile. The KNL processor family has an improved cache and memory organization, where one can choose between five different clustering modes (see `https://colfaxresearch.com/knl-numa/`) and three different memory modes (see `https://colfaxresearch.com/knl-mcdram/`). For all our experiments, we have used the quadrant mode in combination with the cache-mode since this configuration complies to multi-core commodity hardware[67]. Our configuration results in 16 MB shared L3 cache among all cores. We have repeated our synthetic experiments on Intel© Skylake™CPU (see Section 7.5) to demonstrate that our algorithm is superior on a range of different CPU architectures.

Data

In our experiments, we compare *FGF-Hilbert Join* on uniform synthetic data (Table 7.1) as well as on publicly available real data (Table 7.2). In our diagrams, we specify the number of data points n (K for thousands, and M for millions) and the dimensionality d. Throughout all our experiments, we scaled all features to the range between 0 and 1.

Selectivity as Average Number of Neighbors

The standard definition of selectivity [50] of a join operation $\mathbf{X} \bowtie \mathbf{Y}$ is $\textit{selectivity} = \frac{|\mathbf{X} \bowtie \mathbf{Y}|}{|\mathbf{X} \times \mathbf{Y}|}$. However, this measure is not very intuitive for large data sets because although each object may join with a huge number of other objects, the selectivity may appear very small, due to the quadratic growth of the denominator. More intuitive is the average

[1] `https://gitlab.cs.univie.ac.at/martinp16cs/hilbertJoin`

Dataset	*n*	*d*	*Description*
Gaussian	1M	8d	rand. Gaussian distr.
Uniform_200	200K	64d	rand. uniformly distr.
Uniform_600	600K	64d	rand. uniformly distr.
Uniform_select	200K	64d	8 select. dims., (Sec. 7.3)

Table 7.1: Properties of Synthetic Data.

number of join partners per object which is n multiplied with the selectivity:

$$nSelectivity := n \cdot selectivity = \frac{|SimJoin_\epsilon(\mathbf{X})|}{n}.$$

Throughout our experiments, we are interested in an *nSelectivity* between 1 and 100. This means that each point has, on average, between 1 and 100 neighbors in its ϵ-region. We experimentally determine our ϵ-region, to obtain the required *nSelectivity*. We provide *nSelectivity* in our plots with a black dashed line, and its range can be viewed on the right y-axis.

Speedup

Our speedup measure is defined as $\frac{T_{AV}}{T_P}$, where T_{AV} is the time required for the auto-vectorization in a single-threaded environment and T_P is the parallel program we want to measure. Usually, such speedup plots are compared with a diagonal line from the bottom left to the top right representing the ideal speedup. However, our method heavily uses pruning, so we often exceed such ideal speedup curves by a large margin.

Implementation details and Comparison Methods

Our implementation and all competitive methods are implemented in C++ using GNU compiler 6.1.0, OpenMP 4.5 for MIMD and AVX-512 for SIMD parallelism. We used Dimensional Reordering, proposed by [70] as a preprocessing step.

For a fair runtime comparison all algorithms (including SuperEGO) were restricted to the computation of the *self*-join and consider only pairs $(\mathbf{x}_i, \mathbf{x}_j)$ with $i \leq j$. To demonstrate the superiority on the *non-self* join, in Section 7.5 this restriction was removed in all methods. All methods report the *cardinality* of the join result.

Auto-Vectorization. We compare to an auto-vectorized implementation of the nested-loop similarity join, since it gives a solid baseline for multi-core environments.

Dataset	n	d	ϵ
Activity Recognition[118]	33.7M	10d	10^{-4}
BigCross[3, 58]	11.6M	57d	10^{-2}
Higgs[14]	11M	29d	10^{-1}
IoT Botnet[87]	502K	115d	10^{-8}

Table 7.2: Properties of Real Data.

Our auto-vectorized approach is also explicitly parallelized using OpenMP and illustrates the effect of a naive implementation with full advantage of SIMD instructions.

BLAS. Since the major effort of a similarity-join is devoted to the computation of the distances between two join partners $\mathbf{x}_i$ and $\mathbf{x}_j$, it appears reasonable to determine a matrix $\mathbf{M}$ containing the scalar products in its elements $m_{i,j} = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$. Matrix $\mathbf{M}$ is computed through a matrix multiplication of the data matrix and its transpose $\mathbf{M} = \mathbf{X} \cdot \mathbf{X}^\mathsf{T}$. A highly optimized SIMD and MIMD parallel matrix multiplication is available in BLAS (Basic Linear Algebra Subroutines) [47]. To avoid excessive use of memory, we calculate the results block by block with a block size of 5120, which was optimized experimentally. In a MIMD and SIMD parallel algorithm, the entries of $\mathbf{M}$ were processed as in Section 7.4.

SuperEGO has been introduced by Kalashnikov in [69] as an improved version of EGO* [70]. In the multi-core environment this is the most relevant comparison partner. We obtained the code via the author's website[2] which uses *POSIX threads* instead of OpenMP. We modified the recursive calls in the code to consider only those pairs $(\mathbf{x}_i, \mathbf{x}_j)$ with $i \leq j$ to be fair to our competitor.

Detailed Performance Analysis of *FGF-Hilbert Join*

Figure 7.6 analyzes the impact of the various building blocks introduced here in this thesis on the runtime performance. In particular we investigate the impact of the Hilbert-curve (see Section 7.4), the impact of the distance by Scalar product (Section 7.4) and, finally, the full technique *FGF-Hilbert Join*, including the Epsilon Grid Order (Section 7.3).

First, we show the benefit of using a **Hilbert curve** in contrast to the canonical nested loops. Therefore, we have implemented two methods *CANO-Euclid* and *Hilbert-Euclid*. Their only difference is the traversal of the similarity matrix, where *CANO-Euclid*

[2] `https://www.ics.uci.edu/~dvk/code/SuperEGO.html`

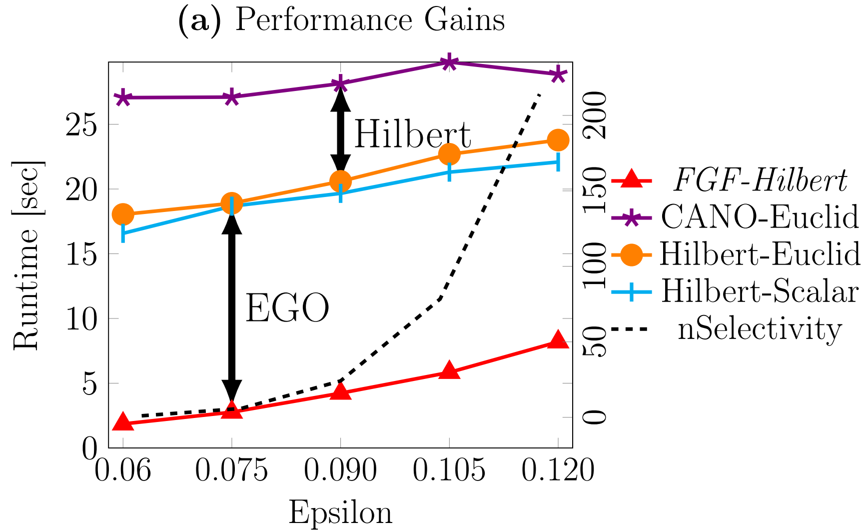


Figure 7.6: The gain in performance of EGO and *FGF-Hilbert Join* .

traverses it line by line, thus in canonical order and *Hilbert-Euclid* uses the traversal by Hilbert order introduced in Section 7.4. For a *nSelectivity* between 1 and 100, we observe a runtime for *Cano-Euclid* between 27.0 and 28.9 seconds, whereas *Hilbert-Euclid* takes only between 18.0 and 23.8 seconds. This is a runtime improvement of a factor between 1.21 and 1.5. We attribute this effect is to the efficient use of the cache hierarchy.

The impact of using different **distance computations** is not very large, but still evident. We compare the two methods *HilbertEuclid* and *Hilbert-Scalar*. They differ in the distance computation, where *Hilbert-Euclid* uses the Euclidean distance and *Hilbert-Scalar* uses the Euclidean distance by scalar product, introduced in Section 7.4. Both approaches iterate over each pair of points and also over the dimensions from 0 to $d-1$. Whenever the squared partial distance already exceeds our ϵ^2, the methods continue with the next pair of points. *Hilbert-Scalar* outperforms *Hilbert-Euclid* in almost all cases. For this dataset, the performance gain is between 1 and 2 seconds, with the exception of $\epsilon = 0.075$, where both approaches are equal. We attribute this performance advantage to a reduced number of numerical operations.

Now, we combine all building blocks with an additional filter step, the **Epsilon Grid Order** (see Section 7.3). It gives our method *FGF-Hilbert Join* an extra boost in terms of runtime efficiency. *FGF-Hilbert Join* takes on this dataset only between 3 and 8 seconds. In comparison to our already SIMD and MIMD effective method *Cano-Euclid*,

this results in a speedup factor between 3.3 and 8.7.

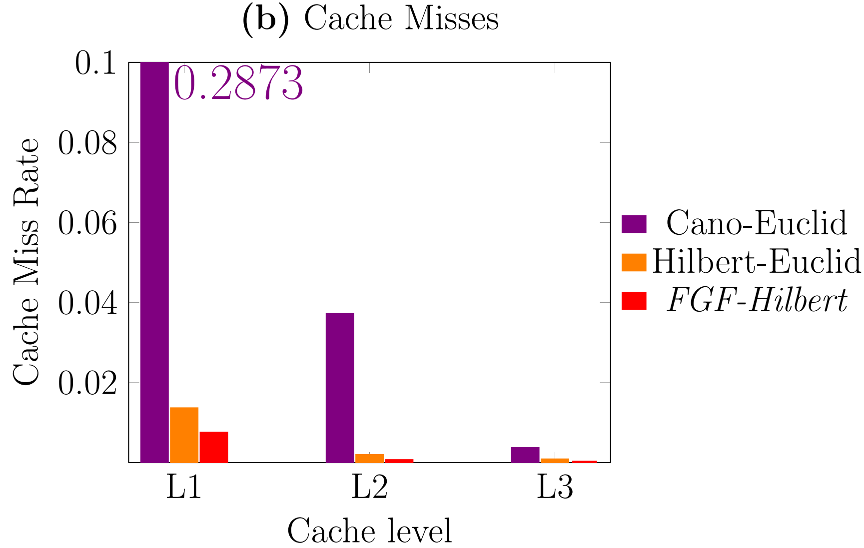


Figure 7.7: Cache misses of Canonical, Hilbert and *FGF-Hilbert Join* .

The **cache miss rate** [67] of *Cano-Euclid*, *Hilbert-Euclid* and *FGF-Hilbert Join* is presented in Figure 7.7. We took the average of the cache miss rates over ϵ from 0.06 to 0.12. The cache misses for *Hilbert-Scalar* are identical to those of *Hilbert-Euclid* and thus not shown. The difference in the runtime between *Cano-Euclid* and *Hilbert-Euclid* is due to the efficient cache use. While *Cano-Euclid* misses 29% of L1, 4% of L2, 0.3% of L3 both the *Hilbert-Euclid* and *FGF-Hilbert Join* are nearly optimal with 0.7% (L1), 0.04% (L2) and 0.002% (L3). There are 41 times fewer L1 accesses and 85 times fewer L2 accesses by the use of the FGF-Hilbert curve!

FGF-Hilbert Join consists of three *individual phases*:

Sorting: EGO-sort is applied to the data (Section 7.4).

Planning: Determining the bounds $lb^{(s)}(i), ub^{(s)}(i)$ for refinement (Section 7.4).

Join: Traversal of the FGF-Hilbert loop (Section 7.4).

Figure 7.8 illustrates the absolute runtime performance on each individual part. For our dataset of one million objects *FGF-Hilbert Join* takes in total 2.200 seconds for the smallest ϵ of 0.06. We sort our data in parallel using OpenMP which takes 0.33 seconds

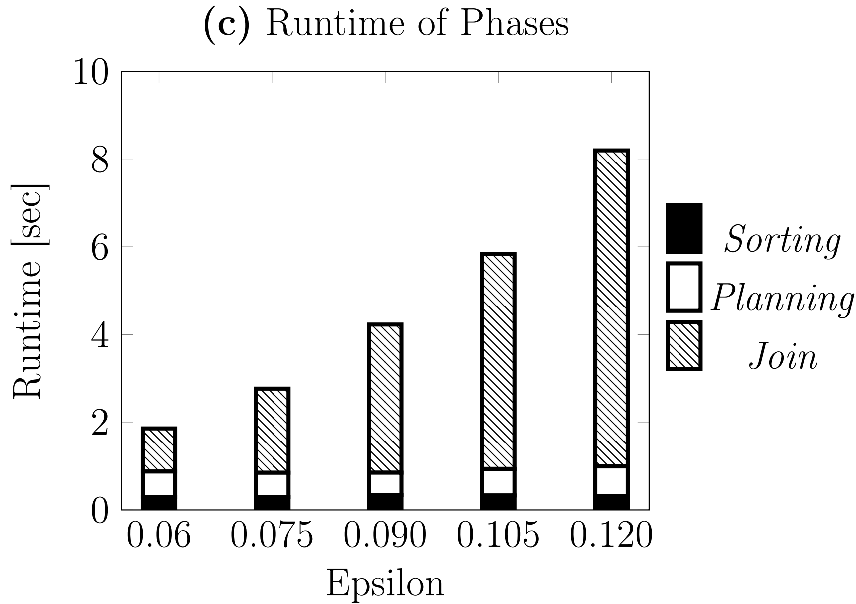


Figure 7.8: Runtime of each phase in *FGF-Hilbert Join* .

or 13% of the total runtime. For the highest *nSelectivity* the sorting rate (as percentage to the full runtime) decreases to 3.7%. Our planning phase takes 0.59 seconds or 27% and the actual join takes 1.28 seconds or 58% of the runtime. For an ϵ of 0.12, where the *nSelectivity* is increased from 1 to 214 the runtime of the join gets increased from 58% to 88% of the total runtime.

Experiments on Synthetic Data

Uniformly Distributed Data.

The challenge on uniformly distributed data is that filter-refinement techniques tend to fail to improve the runtime behavior, due to the "*curse of dimensionality*". Our experiments show, that our technique performs well even on a 64d dataset.

In Figure 7.9, we compare our approach in a uniformly distributed high-dimensional setting. The dataset for Figure 7.9a has 64 dimensions, where each attribute ranges from 0 to 1. The dataset for Figure 7.9b has also 64 dimensions, but only 8 of them range between 0 and 1. To simulate maximum correlation the other 56 attributes share the value of the 8th attribute. This means, the dataset in Figure 7.9b has 8 selective dimensions, where our Epsilon Grid Order (Section 7.3) and our approach (Section 7.4) efficiently

filters out non-join-candidates. In both cases, with and without selective dimensions, our approach is the best choice. The auto-vectorized approach and BLAS are insensitive to variations of ϵ. They have the same runtime behavior of roughly 9 seconds for BLAS and 28 seconds for auto-vectorization. Super-EGO cannot handle such high dimensional and uniformly distributed case (left), but works fine in the selective-dimensional setting (right). Particularly worthy of emphasis is for the completely uniform case in Figure 7.9a, *FGF-Hilbert Join* is still faster than the highly hardware-optimized BLAS variant.

Join with 2 Sets.

The *FGF-Hilbert Join* as a join between 2 sets $\mathbf{X}$ and $\mathbf{Y}$ is outlined in Figure 7.10. Similar to Figure 7.9 we use here 8 selective and 56 non-selective dimensions. For increasing selectivity (Figure 7.10a), *FGF-Hilbert Join* is roughly 7.5 times faster than SuperEGO and twice as fast as BLAS. For different sizes of $\mathbf{X}$ and $\mathbf{Y}$ (Figure 7.10b) *FGF-Hilbert Join* is about 75% faster than BLAS and about 7.5 faster than SuperEGO and between 4 and 5 times faster than auto-vectorization.

Variation of Data Properties.

In Figure 7.11 we demonstrate the variation of different properties like ϵ, dimensionality d, the number of objects n, and the number on threads on uniformly distributed data. In the first plot (Figure 7.11a) *FGF-Hilbert Join* achieves a runtime of 2 seconds for $\epsilon = 0.18$

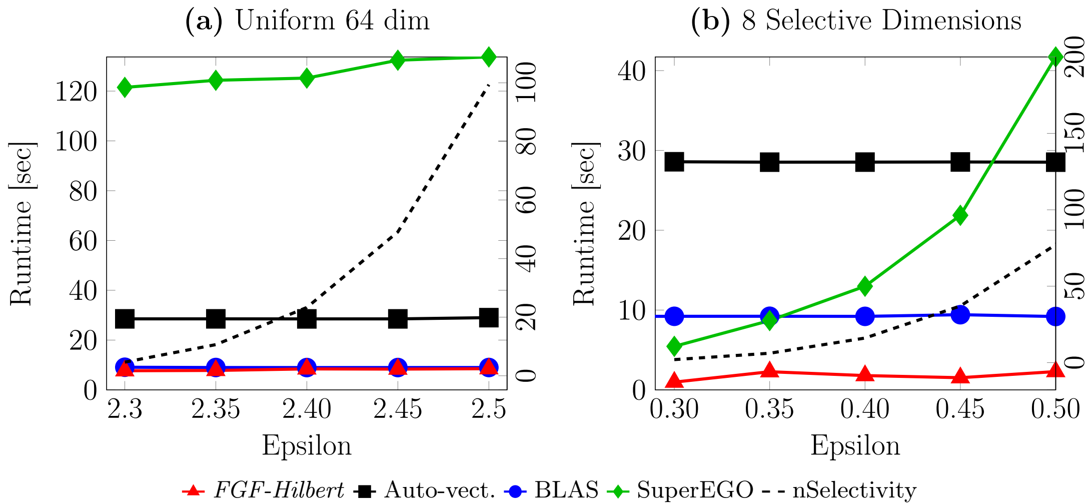


Figure 7.9: (a) Full Uniform. (b) 8 Selective Dimensions and 56 Non-selective Dims. (Uniform, 200K, 64d).

and for $\epsilon = 0.34$ a runtime of 9.13 seconds. *FGF-Hilbert Join* is in every setting at least 10 times faster than Super-EGO and up to 45 times faster than auto-vectorization, as well as 90 times faster than the method using BLAS. In our next plot in Figure 7.11b, we increase the dimensionality up to 32 features. *FGF-Hilbert Join* is in all cases the fastest method. We will see later in real data experiments that *FGF-Hilbert Join* is the fastest method even for very high dimensional cases, like $D = 115$. We also scale across a variation in the number of data points n up to 4 million data points (see Figure 7.11c). In the setting of 4 million data points and $d = 16$, *FGF-Hilbert Join* with a runtime of 518 seceonds is 5.8 times faster than BLAS and 6.3 times faster than auto-vectorization. Super-EGO spent approximately the same time as auto-vectorization and BLAS for only half of the objects.

Gaussian Distributed Data.

Our experiment in Figure 7.12 uses the same setting as described in our experiments on uniformly distributed data. Comparing the two Figures 7.12a and Figure 7.11a, where we keep the *nSelectivity* on the same level, we can see that our *FGF-Hilbert Join* performs similarly in the Gaussian setting as in the uniformly distributed setting. The runtime of *FGF-Hilbert Join* ranges from 2 to 9 seconds. In contrary to our approach, Super-EGO performs in the Gaussian distributed setting worse than in the uniformly distributed setting. The difference in runtime becomes visible with a higher *nSelectivity*, where Super-EGO takes in the uniformly case 119 seconds and in the Gaussian case 138

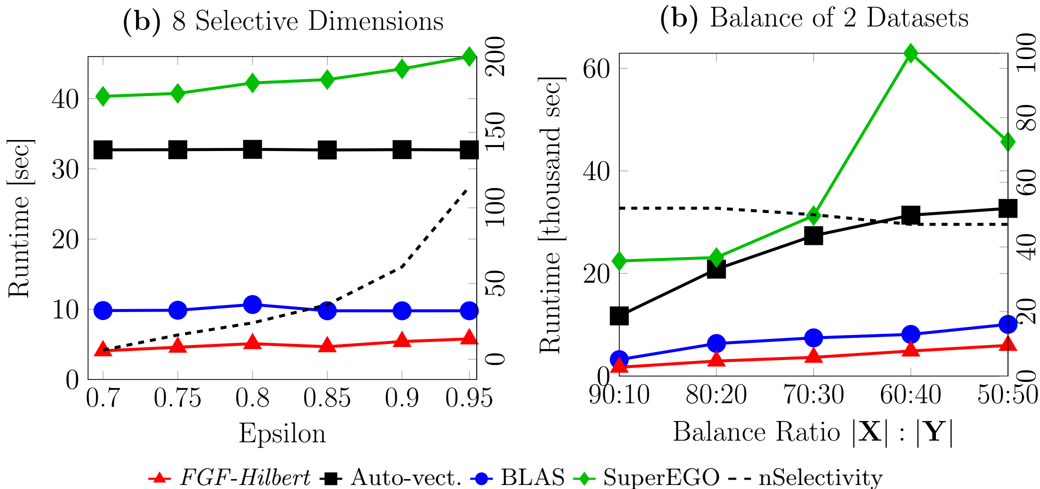


Figure 7.10: Join with Two Sets. 8 Selective Dimensions and 56 Non-selective Dims. (Uniform, $2 \cdot 200K$, 64d).

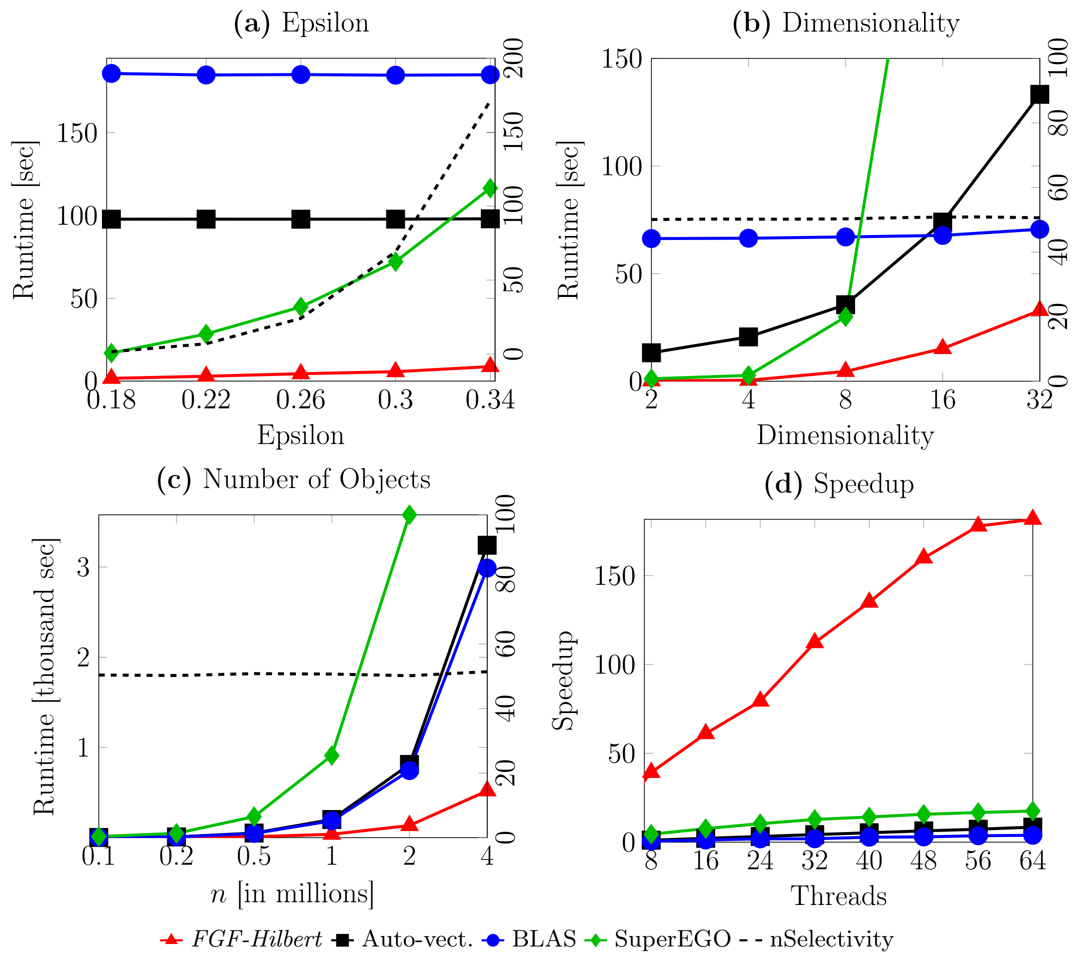


Figure 7.11: Runtime Experiments (Default: Uniformly Distributed, 600K, 8d).

seconds, thus 16% more runtime.

Next, we compare the two different distributions in terms of speedup experiments, where Figure 7.12b describes the Gaussian distributed case and Figure 7.11d shows the uniformly distributed case. We observe a similar runtime behavior.

Experiments on Real Data

Higgs Dataset.

This large dataset makes methods like BLAS or auto-vectorization to an infeasible tool for productive usage. In Figure 7.13a BLAS needs 23 500 seconds and auto-vectorization

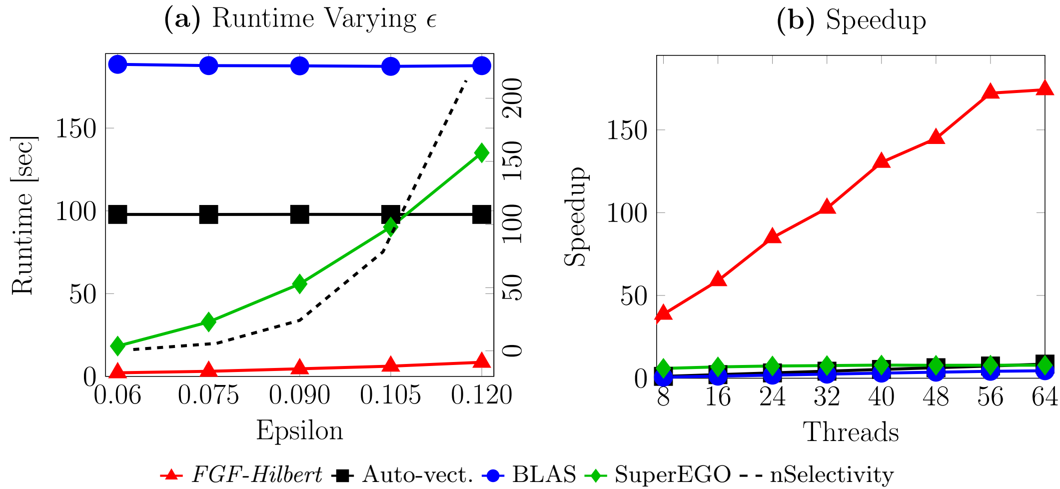


Figure 7.12: Runtime Experiments on Gaussian Data (Gaussian, 600K, 8d).

even 40 400 seconds. We use a dashed line for these two methods, since we could use only a single run for these methods. Throughout our real data experiments, only methods which rely on pruning or effective indexing are of practical use. The runtime of Super-EGO ranges from 3 000 seconds for $\epsilon = 0.3$ to more than 9 400 seconds for $\epsilon = 0.45$. However, *FGF-Hilbert Join* is the fastest method with 290 seconds for $\epsilon = 0.3$ and 540 seconds for $\epsilon = 0.45$.

Activity Recognition Dataset.

In Figure 7.2b our reference implementations without filter and refine techniques, performed poorly. Therefore, we have decided to take them out of our plots. BLAS and auto-vectorization spent more than a day on a single run. For $\epsilon = 2 \cdot 10^{-4}$ *FGF-Hilbert Join* is with 47 seconds 1.77 times faster than Super-EGO which takes 83.7 seconds. For $\epsilon = 12 \cdot 10^{-4}$, where *FGF-Hilbert Join* took 50 seconds in computation, it is 2.3 times faster than Super-EGO.

IoT Botnet Dataset.

The detection of IoT botnet data [87] is a challenge for ϵ approaches, not only because of the high dimensionality, but also in the similarity of the data points. To achieve our desired nSelectivity of roughly 1 to 100 neighbors in the ϵ-region we need an epsilon of 10^{-8}. The runtime of *FGF-Hilbert Join* (see Figure 7.13c) with an average of 2.6 seconds is more than 7.2 times faster than Super-EGO, which has a runtime of roughly 18s.

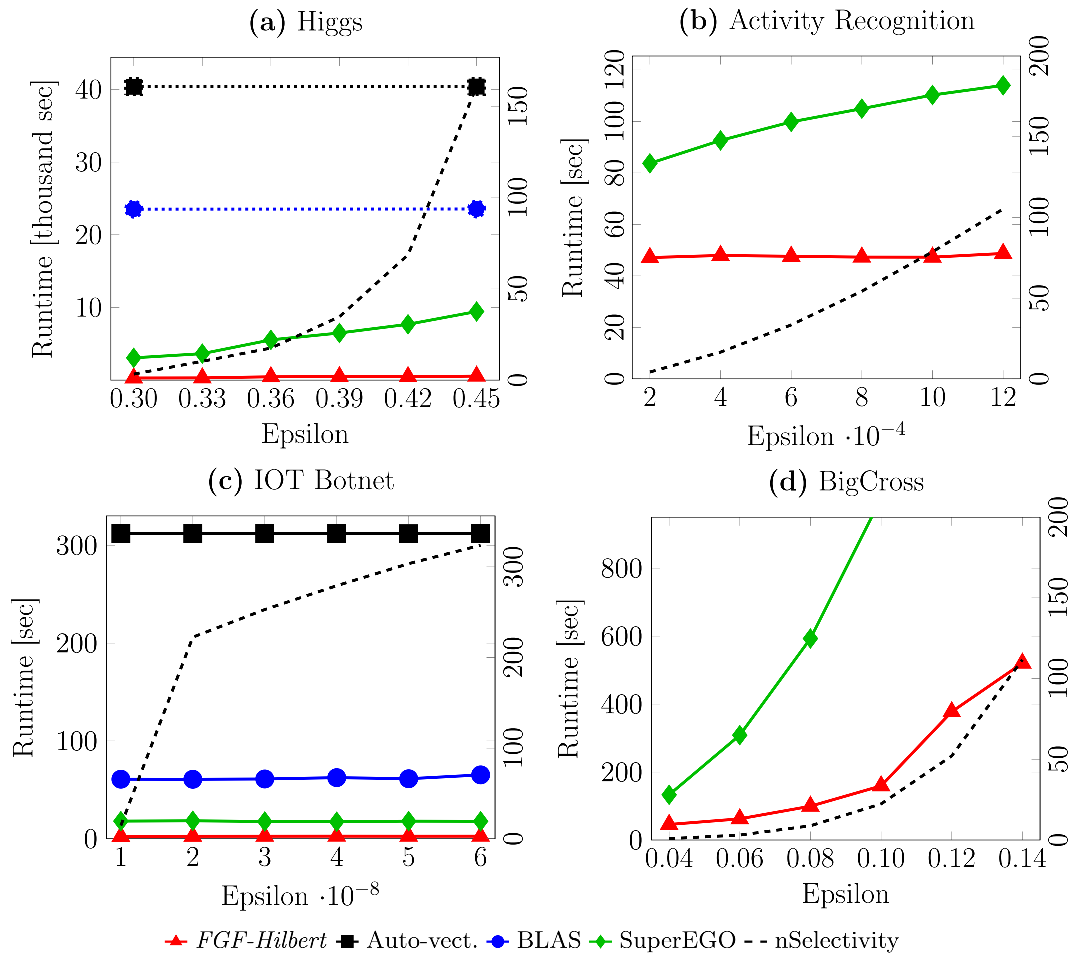


Figure 7.13: Runtime Experiments on Real Data. Properties in Table 7.2

.

BigCross Dataset.

The join algorithm implemented with BLAS and auto-vectorization took more than 8 hours to complete. Super-EGO with a runtime of 133 ($\epsilon = 0.04$) is lagging behind *FGF-Hilbert Join* , which has a runtime of 45 seconds. For a growing nSelectivity of in average 111 neighbours for each point, the gap between those two approaches rises to a factor of up to 4.3, where *FGF-Hilbert Join* has a runtime of 520 and Super-EGO of 2270 seconds, cf. Figure 7.2d.

Experiments on Skylake CPU

Our experiments on Intel® Skylake™CPU[3] show similar behavior like on KNL. In Figure 7.14 we have a similar setting as before in Figure 7.9. On the Skylake CPU, *FGF-Hilbert Join* is still the best choice. Although, the difference between BLAS and *FGF-Hilbert Join* is less evident, we have a larger speedup in comparison to auto-vectorization (from a factor of 3 to 6) and Super-EGO (from 11 to 12) for full uniformly distributed data (Figure 7.14a). For the data with 8 selective dimensions (Figure 7.14b), we are 30% faster than BLAS and outperform auto-vectorization and Super-EGO with a factor of 2 at least.

Figure 7.15 has a similar setting as previously shown in Figure 7.11 with comparable runtime behavior. Our approach outperforms our comparison partners in every aspect.

Speedup experiments on real data

In real data experiments we deal with large datasets and speedup experiments would take quite a long time. For example one single threaded run on the BigCross dataset would take 1 day. The aim of speedup experiments is to visualize scaling among various thread sizes, which is similar if we take a fraction of the data. Therefore, we have decided to use only 10% of the data points for our speedup experiments. The experiments with variations on ϵ are always with the full size and all 64 threads of our machine. For the

[3]Hardware information: `https://tinyurl.com/y8vgn52c`

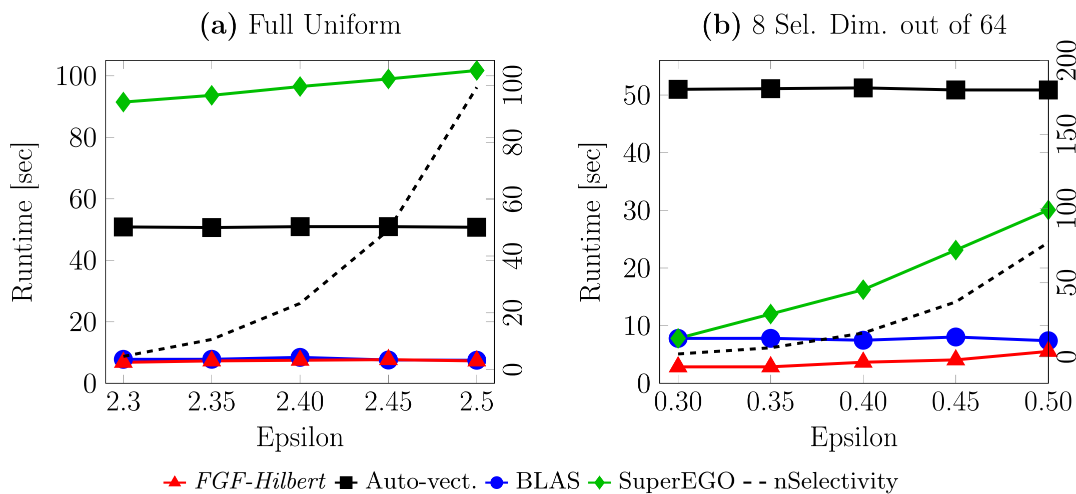


Figure 7.14: Experiments on Skylake CPU (cf. Fig. 7.9)

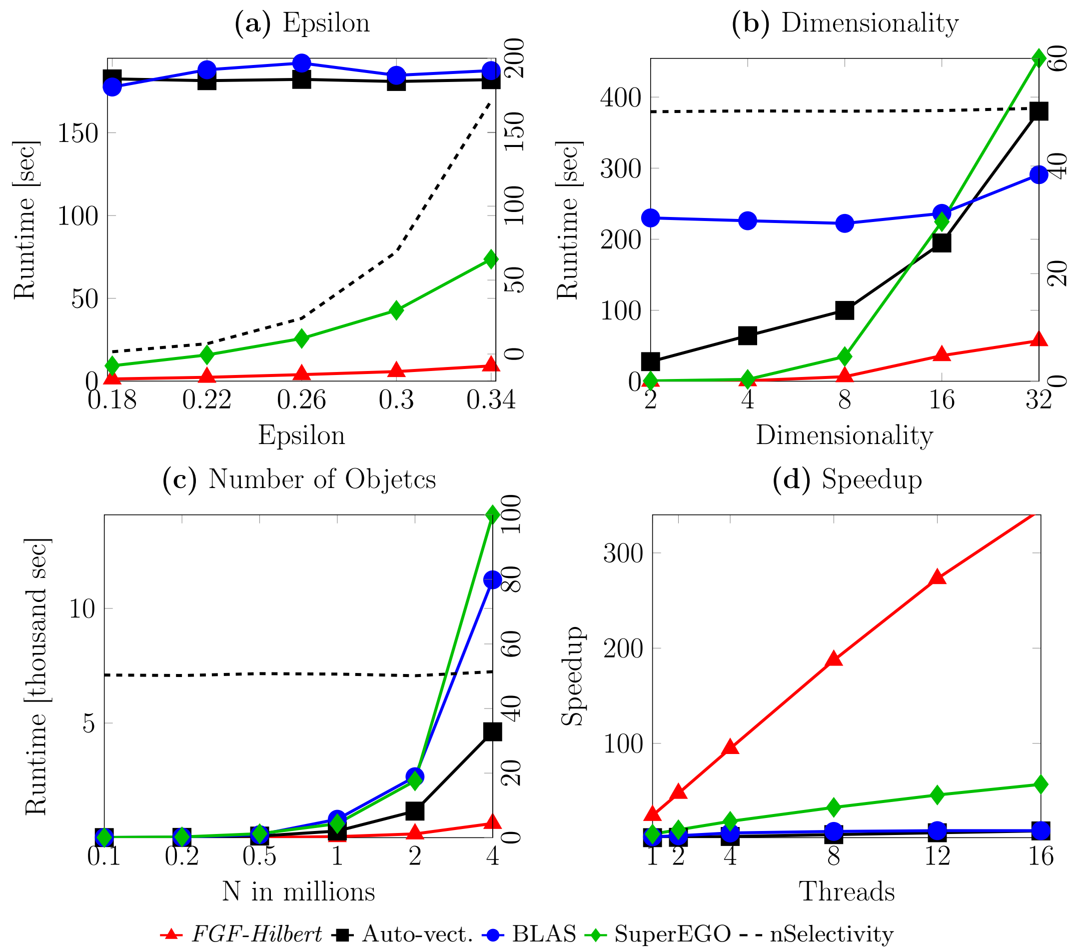


Figure 7.15: Runtime Experiments on Skylake CPU (Default: Uniformly Distributed, 600K, 8d; cf. Figure 7.11).

Higgs dataset (see Figure 7.16a) our approach *FGF-Hilbert Join* is with 7.26 seconds more than 1770 times faster than the single-threaded auto-vectorization approach, which needed 12866.93 seconds. In contrast to that, Super-EGO achieves a speedup of 284 with a runtime of 45.23 seconds.

The most impressive speedup could be achieved with the largest dataset i.e. *Activity recognition* dataset. *FGF-Hilbert Join* is up to 10 000 times faster than the auto-vectorized method.

The dataset *IOT botnet* is by far the smallest dataset but has very high dimensionality. An additional property of this TCP dataset is that the points are relatively dense.

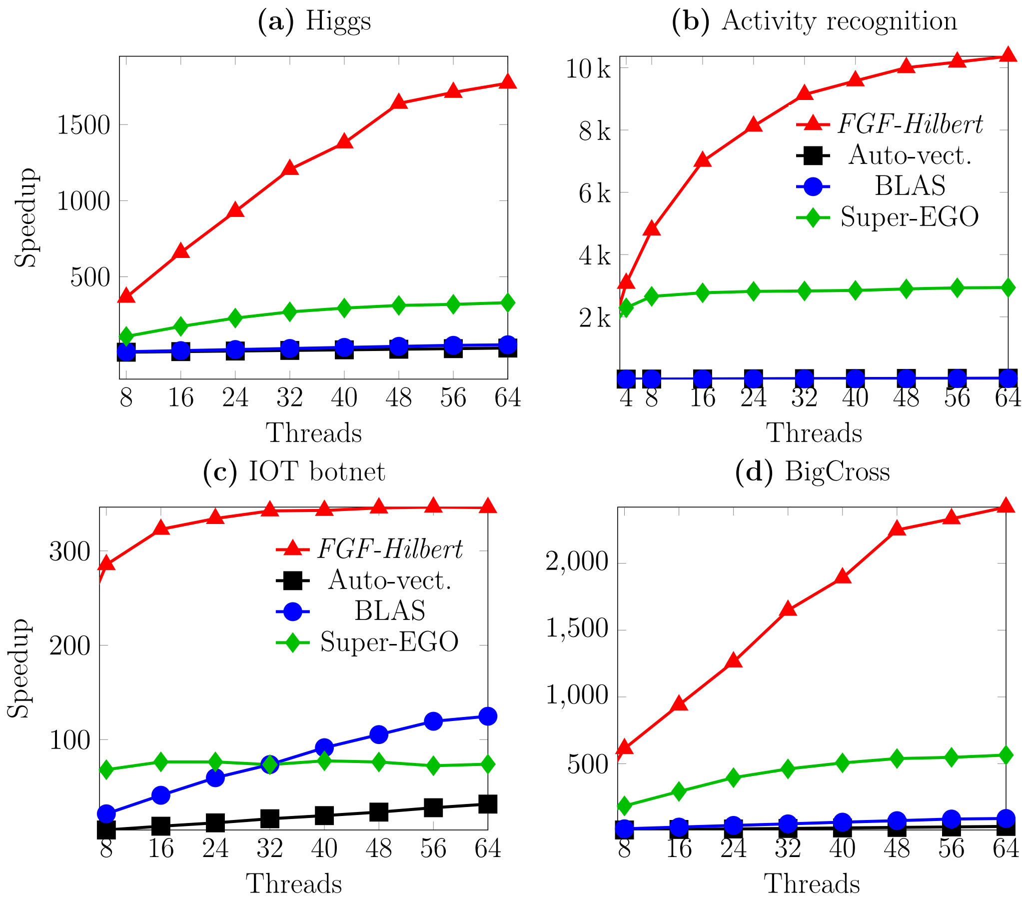


Figure 7.16: Speedup experiments on real data. Properties in Table 7.2.

Unfortunately, Super-EGO failed to compute the dimensional reordering because of the small epsilon of $\epsilon = 10^{-8}$. We decided to turn the dimensional reordering off for this particular dataset. This is why Super-EGO does not finds selective dimensions. Therefore, not much speedup could be achieved on this particular dataset. However, our method spent only 0.3 seconds with the use of 64 threads. This is a speedup of 333. The algorithm spent approximately 60% of its runtime in dimensional reordering, indexing, and sorting. These runtimes explain why we see a flat speedup curve for this experiment.

The *BigCross* dataset shows a similar result, as we have already seen in other datasets, such as *Higgs* or *Activity recognition* dataset. Here, we observe a speedup of 2421, where the runtime of the single-threaded auto-vectorization is 25 889 seconds and our algorithm spent only 10, 69 seconds for the total runtime with 64 threads.

7.6 Related Work and Discussion

The similarity join finds similar pairs of objects within a distance of ϵ. There are various approaches for **different data types**. The data can be of any type, as long as a distance function exists. Fixed-length text data often uses Hamming distance [65] and the similarity between variable length text is often measured using the edit distance [133]. A common measure for set data is the Jaccard distance [43, 134], whereas the similarity of documents is processed with cosine-like similarity measures [8, 114]. In this chapter we focus on double precision vector data for multi-core environments.

Approximate nearest neighbor search techniques can also be applied to the similarity join problem, however without guarantees on completeness and exactness of the result. There may be false positives as well as false negatives. Recently an approach [137] to Locality Sensitive Hashing (LSH) is used on a representative point sample, to reduce the number of lookup operations. LSH is of interest in theoretical foundational work, where a recursive and cache-oblivous LSH approach [95] was proposed. The topic of approximate solutions for the similarity join is also an emerging field in deep learning [97]. There are approximative approaches which target low dimensional cases (spatial joins in 2–3 dimensions [29]) or higher (10–20) dimensional cases [7]. Very high-dimensional cases, with dimensions of 128 and above have been targeted with Symbolic Aggregate approXimation (SAX) techniques [83]) to generate approximate candidates. SAX techniques rely on several indirect parameters like PAA size or the iSAX alphabet size. We consider exact solutions to the similarity join on d-dimensional vectors, where d is typically in the range of $\{2, ..., 64\}$.

There are preconstructed indexing techniques, which are based on **space-filling curves** and applied to the similarity join problem. Specifically, where the data is sorted efficiently with respect to one or more Z-order curves [44, 73, 79] in order to test the intersection of the hypercubes in the datastructures. Others propose space-filling curves, to reduce the storage cost for the index [35]. LESS [79] targets GPUs and not multi-core environments. ZC and MSJ [73] as well as the SPB-tree index [35], although simple, they require space transformations and preprocessing, which make them hard to parallelize. On the contrary, our approach processes the similarity matrix using the Hilbert curve to efficiently exploit the cache hierarchy in modern computer architectures. To the best of our knowledge, this has not been done previously in the context of similarity join algorithms.

Our approach belongs to the **EGO family** of ϵ-join algorithms. The EGO-join algorithm is the first algorithm in this family introduced by Böhm et al. in [19]. The Epsilon Grid Order (EGO) was introduced as a strict order (i.e., an order which is irreflexive, asymmetric, and transitive). It was shown that all join partners of some point $\mathbf{x}$ lie within an ϵ-interval of the Epsilon Grid Order. Algorithms of the EGO family exploit this knowledge for the join operation. The EGO-join has been re-implemented as a recursive variant with additional heuristics to quickly decide whether two sequences are non-join-able [71]. Further improvements proposed two new members of this family, the EGO* [70] algorithm and its extended version called Super-EGO [69] target multi-core environments using a multi-process/multi-thread programming model. Super-EGO proposes a dimensional reordering [69], which has also been applied to our join technique. In our experiments (Section 7.5), Super-EGO encounters some difficulties with uniformly distributed data, particularly when the number of data objects exceeds millions of points or the dimensionality is above 32.

If the similarity join runs multiple times on the same instances of the data, one might consider **index-based approaches** [21, 35, 98], such as R-tree [28] or M-tree [41]. Index-based approaches have the potential to reduce the execution time, since the index stores pre-computed information that significantly reduces query execution time. This pre-computational step could be costly, especially in the case of List of Twin Clusters (LTC) [98], where the algorithm needs to build joint or combined indices for every pair of points in the dataset. The D-Index [45] and its extensions (i.e. eD-Index [46] or i-Sim index [102]) build a hierarchical structure of index levels, where each level is organized into separable buckets and an exclusion set. The most important drawback of D-Index, eD-Index and i-Sim is that they may require rebuilding the index structure for different ϵ.

Data partitioning across multiple machines is out of the scope of our method. Our assumption is based on shared memory envirnoments, where we assume that the data fits into main memory. The case of relational join algorithms has been studied extensively in the past [52, 110, 125]. The similarity join has been successfully applied in the distributed environment with different MapReduce variants [52, 76, 85]. Another distributed version is proposed in [140]. There, a multi-node solution with load-balancing is used, that does not require re-partitioning on the input data. This variant focuses on minimization of data transfer, network congestion and load-balancing across multiple nodes.

The similarity join has been already implemented for **Graphics Processing Units (GPUs)**. In [22] the authors use a directory structure to generate candidate points. On datasets with 8 million points, the proposed GPU algorithm is faster than its CPU variant, when the ϵ-region has at least 1 or 2 average neighbors. LSS [79] is another similarity join variant for the GPU, which is suited for high dimensional data. Unfortunetly both [79] and [22] are targeted to NVIDIA GPUs and have been optimized for an older version of CUDA.

Cache-oblivious Algorithms

Cache-oblivious algorithms [55] have attracted considerable attention as they are portable to almost all environments and architectures. Algorithms and data structures for basic tasks like sorting, searching, or query processing [60] and for specialized tasks like ray reordering [88] or homology search in bioinformatics [51] have been proposed. Two important algorithmic concepts of cache-oblivious algorithms are localized memory access and divide-and-conquer. Our Hilbert curve integrates both ideas. The Hilbert curve defines a 1D ordering of the points of an 2-dimensional space such that each point is visited once. Most related to our work, Bader et al. proposed to use the Peano curve for matrix multiplication and LU-decomposition [9, 12]. The algorithms process input matrices in a block-wise and recursive fashion where the Peano curve guides the processing order and thus the memory access pattern. In [23], cache-oblivious loops have been applied to K-means clustering and matrix multiplication. We considerably improve memory locality and runtime by introducing the Fast General Form (FGF-) Hilbert curve (see Section 7.4).

Optimized Techniques for Specific Tasks or Hardware

The library BLAS (Basic Linear Algebra Subprograms) [47] provides basic linear algebra operations together with programming interfaces to C and Fortran. BLAS is highly hardware optimized: specific implementations for various infrastructures are available, e.g. ACML for AMD Opteron processors or CUBLAS for NVIDIA GPUs. The Math Kernel Library (MKL) contains highly vectorized math processing routines for Intel processors. In contrast to our work, these implementations are very hardware-specific and mostly vendor-optimized. Moreover, they are designed to efficiently support specific linear algebra operations. Our experiments demonstrate that our cache-oblivious approach reaches a performance better than BLAS on the task of the similarity join for points of dimensions in the range of $\{2, ..., 64\}$.

7.7 Conclusion

Databases running in-memory on multi-core microarchitectures depend upon the efficient usage of all levels of the cache hierarchy for high performance. The *FGF-Hilbert Join* optimally exploits the complete cache hierarchy ranging from super-fast registers to the relatively slower L3 cache by processing potential join partners in the order defined by the FGF-Hilbert curve, which is a novel space-filling curve. This curve inherits from the classical Hilbert curve its optimal locality but allows traversing regions of arbitrary size and shape. *FGF-Hilbert Join* integrates this idea into a filter-refinement algorithm. After the filter step, potential candidates are efficiently refined in FGF-Hilbert order by multiple threads in parallel using highly efficient SIMD parallelism for the distance calculations. Experiments demonstrate that FGF-Hilbert Join outperforms comparison methods designed for disk-based systems. In future work, we want to consider similarity joins on general metric spaces, with k-nearest neighbor predicates, multi-way joins, and other database operators.

Chapter 8

Cache-oblivious Morton-order Curve

8.1 Introduction

The LU decomposition (or factorization) is an essential element used in many linear algebra applications. Furthermore, it is used in LINPACK to benchmark the performance of modern multi-core processor environments. The LU decomposition algorithm is used in many linear algebra applications, such as solving linear equations of the form $Ax = b$. Solving such systems with linear equations in A with multiple right-hand vectors x is a common task for all kinds of scientific fields and computer-aided workloads. The LU decomposition decomposes the matrix A into two factors - a lower triangular matrix L and an upper triangular matrix U:

$$A = LU \tag{8.1}$$

The advantage of breaking up one linear set A into two successive ones L, U is that the solution of the triangular set of equations is relatively trivial by first solving for the auxiliary vector y, where $L \cdot y = b$ by forwarding substitution and then solving $U \cdot x = y$ for x by a backward substitution.

In previous chapters, we applied the Hilbert curve to the various algorithms, but unfortunately, the Hilbert curve is not applicable in the case of the LU decomposition. The LU decomposition and many other algorithms, such as Crout algorithm [107], the forward- and backward substitution process, or dynamic programming approaches such as the Needleman-Wunsch algorithm [91] have data dependencies to previously calculated

entries in the matrix. The current cell accesses cells to the top and to the left for its computation. In other words, before visiting a cell in a matrix, all entries to the top and to the left must have been precomputed. Throughout this chapter, we call this data dependency issue a monotonicity property. This monotonicity property are fulfilled by the Morton order curve (Z-order and И-order), but not by the Peano- or Hilbert curve (cf. red lines in Figure 8.1).

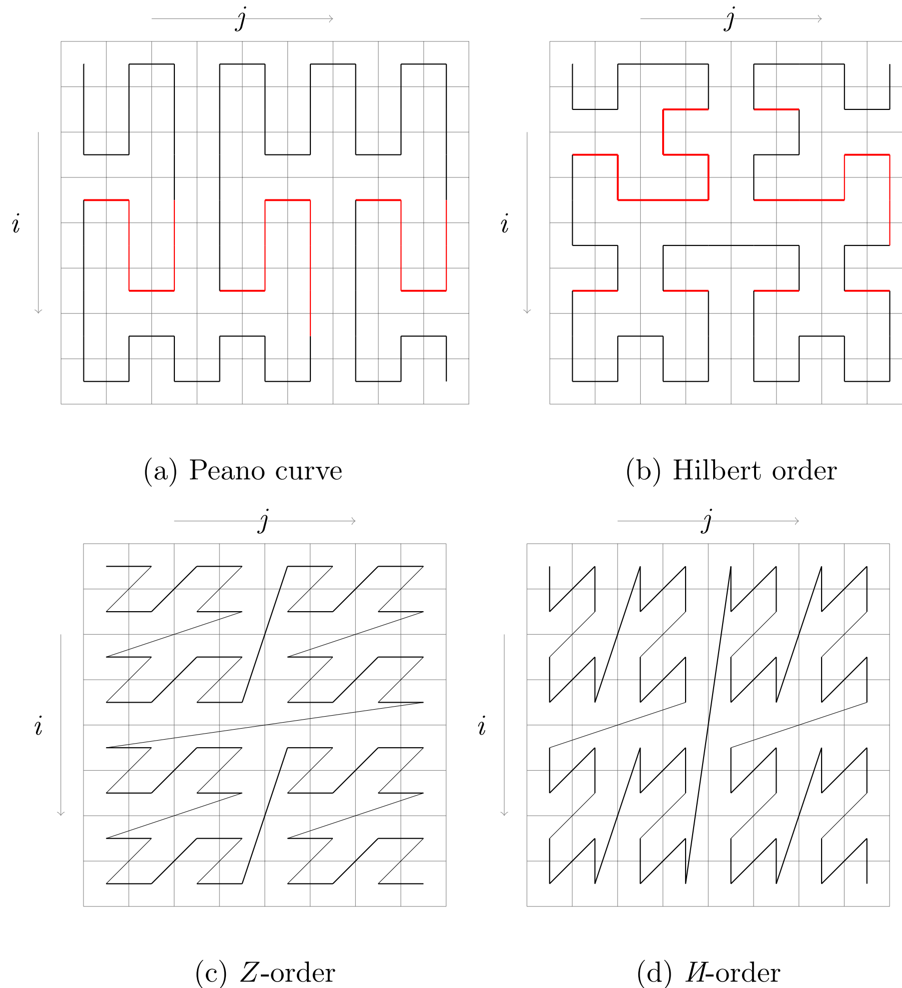


Figure 8.1: Strategies for space-filling curves, violations of monotonicity properties marked in red.

This chapter is organized as follows: in section 8.2 we introduce two different Morton-order generation approaches. One is based on pattern extraction (pext), and the other is based on counting trailing zeros (tzcnt). Both generation approaches have the major

drawback that they are only applicable for $n \times n$ matrices, where n is limited to a power of 2. We address this limitation and extend the traversal to arbitrary $n \times m$ matrices with a concept called microcells. We outline our parallelization for MIMD and SIMD in section 8.3 and evaluate our Morton-order traversal combined with microcells, for example, algorithms such as matrix-multiplication, the forward and backward substitution process, and to the LU decomposition in section 8.4. In section 8.5 we evaluate our different Morton-order approaches with experiments on two different hardware settings for Multi- and Many-core microarchitectures. We give a detailed discussion in section 8.6 and conclude in section 8.7.

8.2 Generating Morton-order Curves

Here we show two different approaches to generate the traversal in Morton-order. The traditional approach is based on pattern extracting (pext) of interleaving bits, and a novel approach based on counting the trailing zeros (tzcnt). Both approaches reverse the process of bit-interleaving. Its time complexity is linear in the number of bits of m, which means $O(\log m)$. However, the coordinate pairs assigned to subsequent order values of m and $m+1$ are not entirely independent. Therefore, we do not need the whole $O(\log m)$ process of reverse bit interleaving. We introduce microcells to eliminate well-known limitations to loop bounds of the power of 2. Both approaches (pext and tzcnt) are then combined with microcells to get later evaluated in the experiments section 8.5. A common way of handling arbitrary-sized matrices is to apply "padding", such that the padded matrix can be subjected to Morton-ordering [122]. As we will see in our experiments, both approaches based on microcells are roughly 5% faster than the approach based on padding.

Traditional Generation with Dilated Integers

The Morton-order curve has been introduced by Morton et. al. in 1966 [89] as an effective approach to access files on the example of geodetic databases. An effective way to generate Morton-order curves using dilated integers have been proposed by Wiese et. al. [130]. The main idea is to derive the (i, j) pair from a Z-order value m, where m is between 0 and 2^t. Let $m = Morton_{pext}(i, j)$ be the function which determines for a coordinate pair $(i, j) \in \mathbb{N}^2$ the Morton-order value m and $(i, j) = Morton^{-1}_{pext}(z)$ be its inverse. The sequence of every even bit from the binary representation from z forms its i-dilation and the sequence of every odd bit builds its j-dilation. This is depicted in

Figure 8.2, where the the Morton-order value z is colored black in the decimal format with its binary representation to the right. The corresponding i and j values are depicted in blue and red.

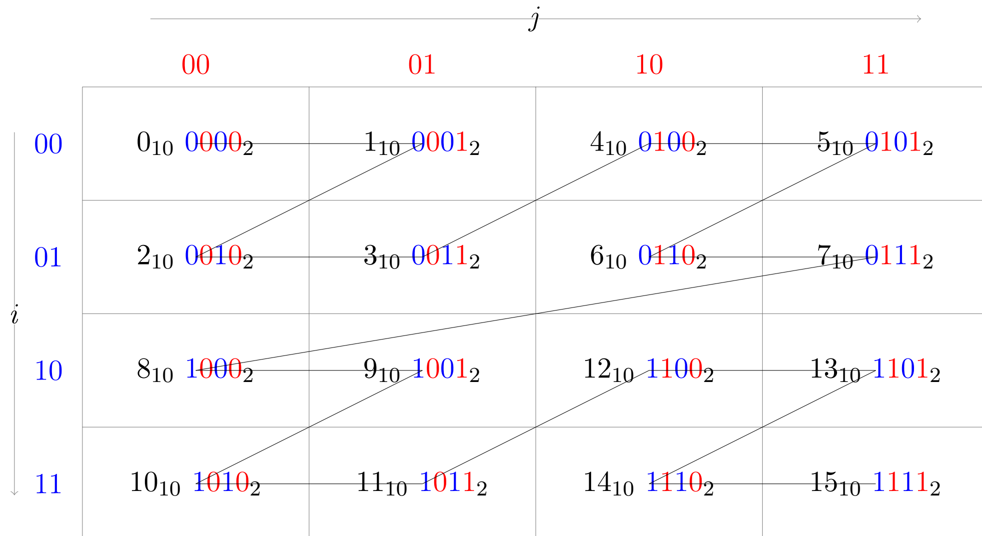


Figure 8.2: Morton-order. Interleaving the binary coordinates from i and j yields the binary z-values shown.

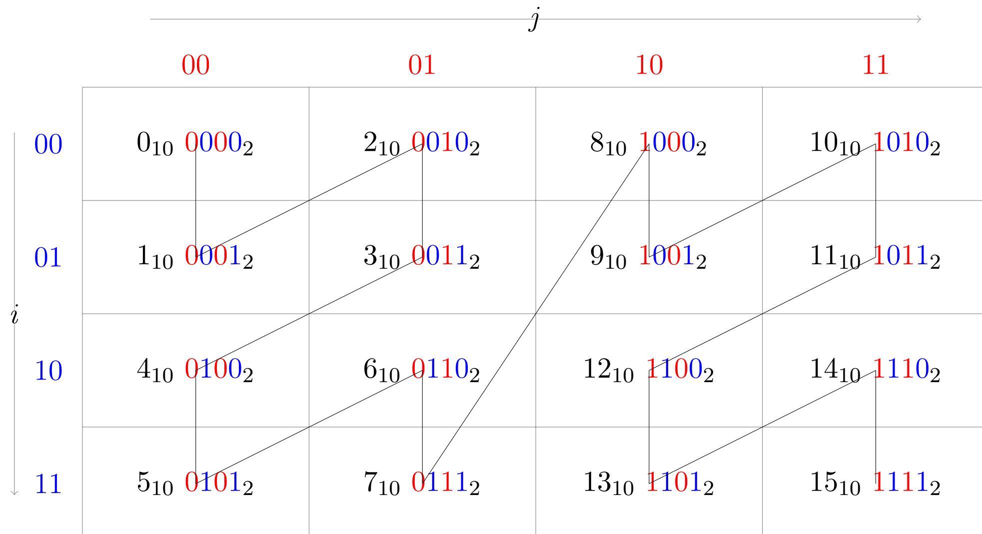


Figure 8.3: И-order. Interleaving the binary coordinates from i and j yields the binary z-values shown.

As a simple example for the Z-order in Figure 8.2, consider the value 6 in decimal

system denoted as 6_{10}, which is 0110_2 in its binary representation. The i-dilation of 6_{10} is 01_2 (1_{10}) and its j-dilation results into a j-value of 10_2 (2_{10}). On today's hardware this can be effectively computed using the instrinsic function _pext_u64 (m) (c.f. Algorithm 4). The variables *EVEN* and *ODD* are masks in binary representation, where every even or odd bit is set. The function _pext_u64(mask, m) extracts bits from the unsigned 64-bit integer m at the corresponding bit locations specified by the masks *EVEN* and *ODD*. The actual loop body which performs the productive work on the generated pairs (i, j) is indicated by the placeholder "*process loop body*" at line 7. In the case of matrix multiplication, this would be the scalar product of the ith row of the matrix A with the jth column of the matrix B and store it in $C_{i,j}$.

Algorithm 4 goes through the cells of the matrix in a Z-order, but it can be easily adapted to the case of the И-order by exchanging the EVEN and ODD parameters for i and j respectively at the *_pext_u64* function (c.f. Figure 8.3).

Algorithm 4 MORTON-ORDER (pext) loop

Input: $t \in \mathbb{N}$
Output: i, j in Morton order

1: $(i, j) \leftarrow 0$
2: $EVEN \leftarrow 0xaaaaaaaa$
3: $ODD \leftarrow 0x55555555$
4: **for** $m \leftarrow 0$ to $2^t - 1$ **do**
5: $\quad$ $i \leftarrow$ *_pext_u64(m, EVEN)*
6: $\quad$ $j \leftarrow$ *_pext_u64(m, ODD)*
7: $\quad$ *process loop body (i,j)*
8: **end for**

Generation of Morton Loops by Counting Trailing Zeros

Here, we present an alternative way to generate the Morton order curve by counting the trailing zeros (tzcnt) from the binary representation of the control variable. Let $m = Morton_{tzcnt}(i, j)$ be the function which determines for a coordinate pair $(i, j) \in \mathbb{N}^2$ the Morton-order value m and $(i, j) = Morton^{-1}_{tzcnt}(m)$ be its inverse. We propose here a novel approach for a loop generating the pairs (i, j) in Morton order which is non-recursive and has only a constant time complexity per loop iteration. We exploit the general observation that when proceeding from m to $m + 1$, one of the variables i or j is increased by one, and in the other, some trailing bits are set to zero (cf. Figure 8.4 for the Z-order and Figure 8.5 for the И-order). Both the selection of the variable and the number of bits depends on the number a of trailing bits that the Morton order

value m has. This number can be determined as $a := \log_2(m \textbf{ and}_{\text{bitw}} (-m))$, where $-m$ is the two-complement. This operation of counting the trailing bits is equivalent to the operation $a :=$ _tzcnt_u64 (m), which is available in hardware on most of today's processors. If a is **odd**, we increase i and reset j. If a is **even**, we increase j and reset i. Due to the Morton order's recursive structure, the number of bits to be reset is $\lceil \frac{a}{2} \rceil$ which is rounded up explicitly in the following pseudo-code segment. A bitwise **and performs the reset** and a shift-left operation noted by $\ll$.

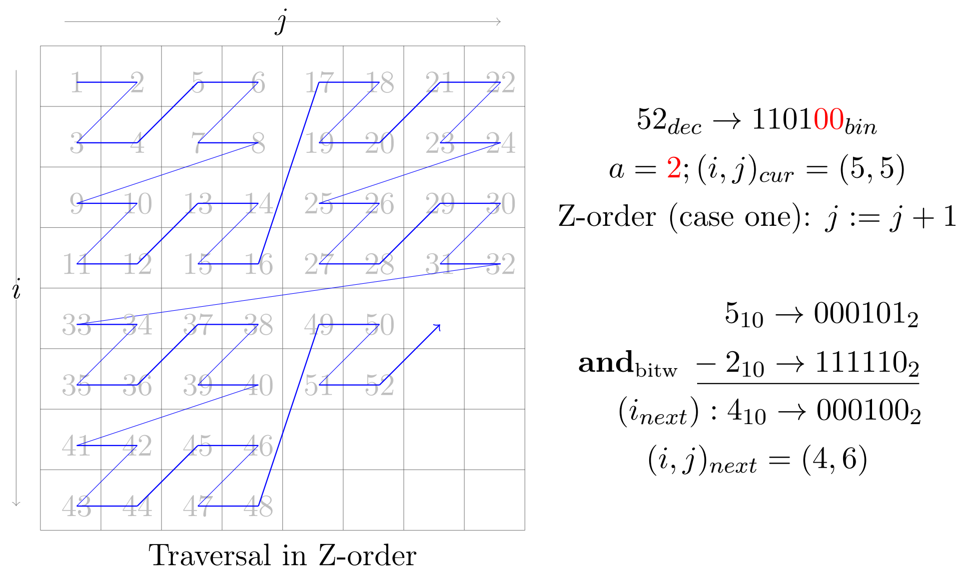


Figure 8.4: Z-order traversal (tzcnt).

Microcells

The two basic algorithms (pext and tzcnt), introduced in the previous section, generate the sequence of all $(i,j) \in \{0,...,n-1\}^2 \subseteq \mathbb{N}_0^2$ if n is a power of two. We improve this basic algorithm for the more general case in which n is not a power of two, and that i and j have different upper and lower boundaries. Firstly, if the range of i-values is more than twice the range of j-values, we stack two or more independent Morton curves above each other (c.f. Algorithm 6 loop at line 13). Analogously, if the range of j-values is more than twice the range of i values, we put independent curves side by side. This case is illustrated in Figure 8.7. For the case, where i and j are within the same power of two, the independent curves have then the property that $\lceil \log_2(i_{\max} - i_{\min}) \rceil = \lceil \log_2(j_{\max} - j_{\min}) \rceil (=: \nu)$, i.e. they are fairly square-like (c.f. Algorithm 6 loop at line 20). The remaining asymmetry

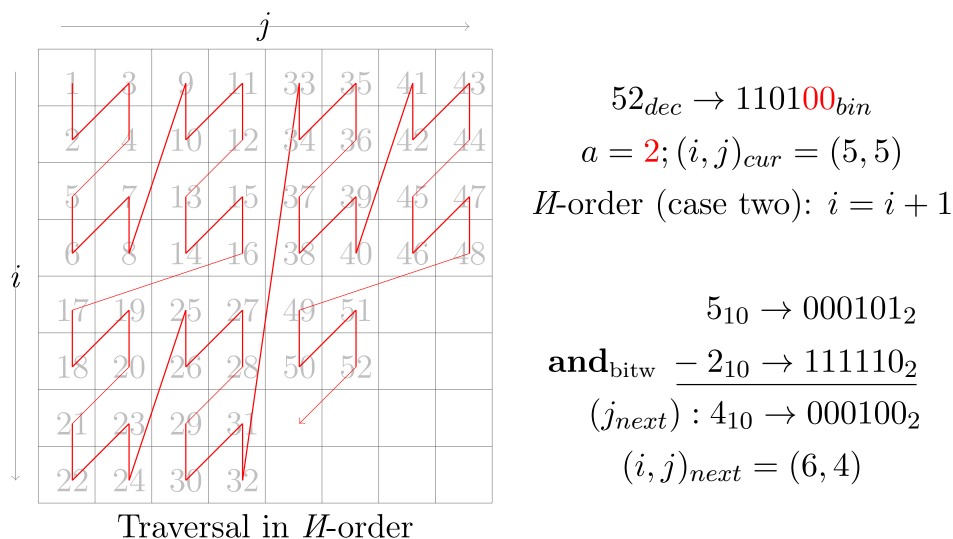


Figure 8.5: И-order traversal (tzcnt).

Code	*Value*
0	stop
1	n.a.
2	-3
3	-2
4	-1
5	0
6	+1
7	n.a.

Table 8.1: Code table for processing microcells

and deviation of the side lengths from being a power of two is then handled by a concept of microcells (c.f. Algorithm 6 loop at line 27).

Microcells enable the use of arbitrary loop bounds, which do not correspond to a power of 2. We disassemble any grid of size $n \times m$ into a number of sub-grids, each having size $r \times s = \{2, 3, 4\} \times \{2, 3, 4\}$. We generate a simple Morton loop with $n = 2^{\lfloor \log_2(\mathrm{n}) \rfloor - 1}$ and fill every generated pair with a sub-grid of size $\{2{\times}2\}$, $\{2{\times}3\}$, $\{2{\times}4\}$, $\{3{\times}3\}$, $\{3{\times}4\}$, or $\{4 \times 4\}$, each obeying the monotonicity properties of Morton order (cf. Figure 8.6). Any grid of any size $n_i \times n_j$ can be generated in this way, and the resulting loop generates every of these $n_i \cdot n_j$ pairs at constant time and space.

Algorithm 5 MORTON-ORDER (tzcnt) loop

Input: $n \in \mathbb{N}$
Output: i, j in Morton order

1: $(i, j) \leftarrow 0$
2: **for** $m \leftarrow 1$ to n^2 **do**
3:     *process loop body* (i, j)
4:     // calculate (i, j) for next iteration:
5:     $a \leftarrow$ _tzcnt_u64(m) //count trailing zero bits
6:     **if** a mod $2 = 1$ // И-order a mod $2 = 0$ **then**
7:         //case one
8:         $i \leftarrow i + 1$
9:         $j \leftarrow j$ $\textbf{and}_{\text{bitw}}$ $(-1 \ll \frac{a+1}{2})$; // reset $\lceil \frac{a}{2} \rceil$ bits
10:     **else**
11:         //case two
12:         $j \leftarrow j + 1$
13:         $i \leftarrow i$ $\textbf{and}_{\text{bitw}}$ $(-1 \ll \frac{a}{2})$; // reset $\lceil \frac{a}{2} \rceil$ bits
14:     **end if**
15: **end for**

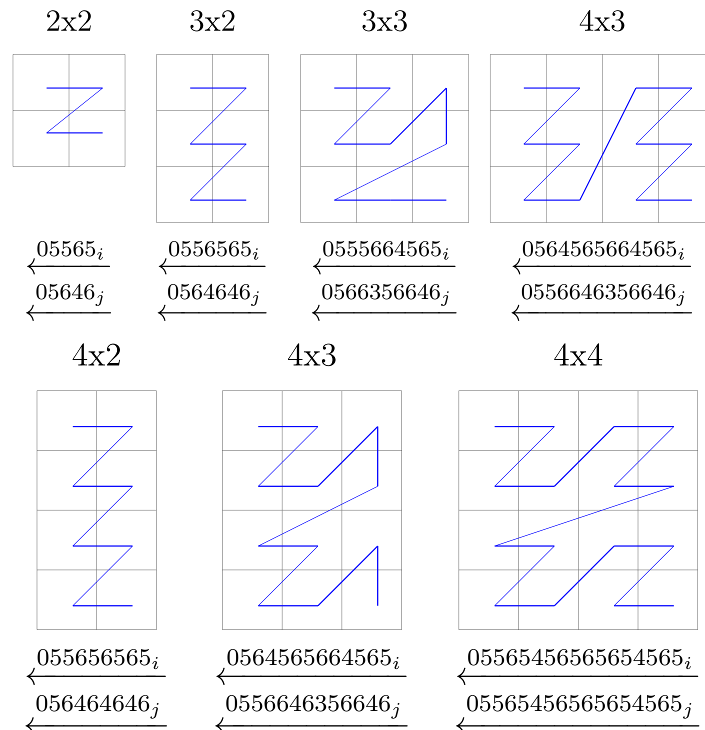
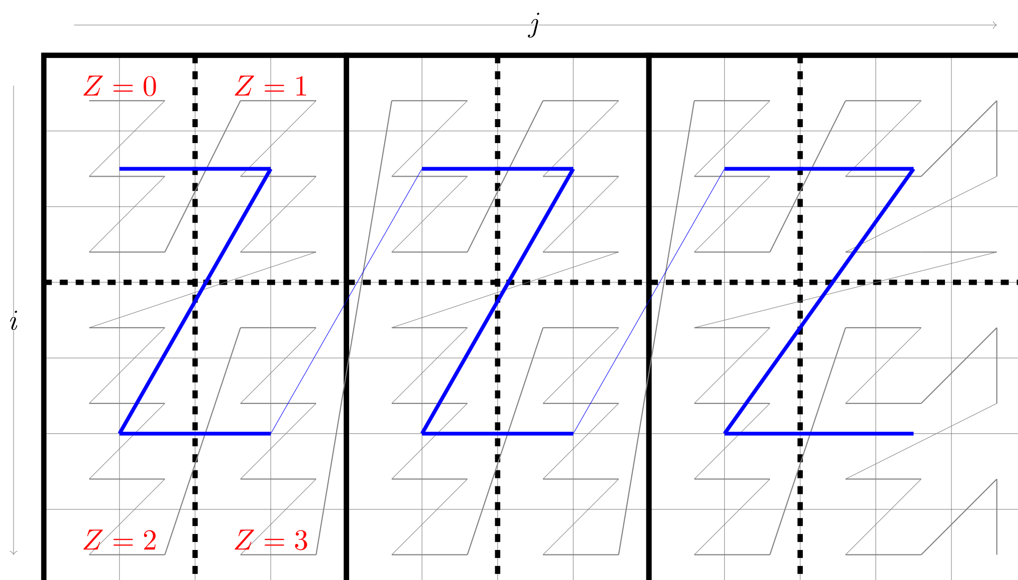


Figure 8.6: Microcell templates



Microcell placement in a 7x13 grid

Figure 8.7: Microcell placement of 5 $\{2 \times 3\}$, 1 $\{3 \times 3\}$, 5 $\{2 \times 4\}$ and 1 $\{3 \times 4\}$ templates (c.f. Figure 8.6) into a 13x7 grid.

The implementation is summarized in Algorithm 6 and here we accompany this algorihtm by an example which is depicted in Figure 8.7. In this example i ranges from 0 to 7 and j from 0 to 13. At first, we divide the problem of arbitrary loop bounds into a problem, where m and n are possibly different, but in the same power of two: $t := \lfloor \log_2(\mathrm{n}) \rfloor = \lfloor \log_2(\mathrm{m}) \rfloor$. In our example ($t = 2$) these are the three thickly bordered grids, where in the first one i ranges from 0 to 7 and j ranges from 0 to 4. We can place several Morton orders of the size 2^t (2^2) side by side, where the first one has the width of 2^t, which corresponds to the *stepsize* in Algorithm 6 at line number 12. In other words, in the first loop we loop over the thick grid and in the second loop (at line 20) we iterate over microcells. The variables I and J represent the offsets for the cells in this grid. In this example, the variable J will become $\{0, 2, 4\}$, and I remains at zero. Each cell has a $m \times n$ subgrid, where we can span a Morton-order with the size of $2^{t-1} \times 2^{t-1}$ over the microcells, where Z ranges from 0 to 3. In each iteration we determine the size of the next microcell template (*idiff* and *jdiff*), which will be processed in the next loop at line 27. Our microcells are small templates, containing lookup tables to calculate the absolute differences from one iteration to the next. A microcell template is a bit-sequence which can be stored in an integer variable. The bit-sequence contains codes that store

the difference to calculate the current i and j variables based on the previous ones. The codes are summarized in Table 8.1 and they can be easily accessed using an $\mathbf{and}_{\text{bitw}}$ operation, which extracts the last 3 bits from the current microcell template. The microcell template reads from right to left, and therefore, the difference can be extracted and stored in a temporary variable. The remaining part of the algorithm is used to ensure the Morton order placement of the microcells (c.f. blue line in Figure 8.7) and has been explained in Algorithm 4.

The user can freely choose the names of iterator variables i and j and can also define local variables inside and outside the Morton loop's loop body. The compiler has all the options to optimize the generated code because all functionality is in a single method.

Architecture and Usability

Our algorithm is non-recursive, and it removes all restrictions on loop boundaries for i and j. The resulting Morton-loop has been implemented as a preprocessor-macro in C/C++, which facilitates its usage in any given algorithm. We will note our cache-oblivious loops following the Morton-order curve in our pseudo-codes as follows:

MORTON-ORDER $(i, j) \in \{0, \ldots, imax - 1\} \times \{0, \ldots, jmax - 1\}$ **do**
  process loop body (i,j)
END-MORTON-ORDER

In this example i ranges from 0 to $imax$ and $j - 1$ ranges from 0 to $jmax - 1$. This case is very analogous to in our source-code for C/C++ where we define our preprocessor macro:

int i,j;
ZORDLOOP_START $(i, j, imin, imax, jmin, jmax)\{$ *printf("%d, %d",i,j);*
$\}$**ZORDLOOP_END** (i, j)

In the literature, there is often a recommendation to use iterators for such applications, but iterators are very specific in how they are used for their application. Iterators are bound to specific pointers in matrices, and in contrast to this, macros are much more generic because they generate loop indices. The Morton-order can also be used in indexing techniques, where the Morton-order helps resolve the index structure. This

Algorithm 6 Microcell placement with Morton-order (pext).
Implemented as a preprocessor macro.

Input: $imin, imax, jmin, jmax \in \mathbb{N}$
Output: i, j in Morton order

1: $EVEN \leftarrow 0xaaaaaaaa$
2: $ODD \leftarrow 0x55555555$
3: $(i, j) \leftarrow (imin, jmin)$
4: $(irange, jrange) \leftarrow (imax - imin, jmax - jmin)$
5: **if** irange < jrange **then**
6: $\quad t \leftarrow \lfloor \log_2(\text{irange}) \rfloor$
7: $\quad kbound \leftarrow (jrange/t/2) * e$ // where $kbound \geq 1$
8: **else**
9: $\quad t \leftarrow \lfloor \log_2(\text{jrange}) \rfloor$
10: $\quad kbound \leftarrow (irange/t/2) * e$ // where $kbound \geq 1$
11: **end if**
12: $stepsize \leftarrow 2^{t-1}$
13: **for** $k \leftarrow 0$ **to** $kbound$ **by** $stepsize$ **do**
14: $\quad$ **if** $irange < jrange$ **then**
15: $\quad\quad I \leftarrow imin,\ J \leftarrow k + jmin$
16: $\quad$ **else**
17: $\quad\quad I \leftarrow k + imin,\ J \leftarrow jmin$
18: $\quad$ **end if**
19: $\quad Z \leftarrow 0$
20: $\quad$ **while** $Z < stepsize^2$ **do**
21: $\quad\quad II \leftarrow I+$ _pext_u64(Z, EVEN)
22: $\quad\quad JJ \leftarrow J+$ _pext_u64(Z, ODD)
23: $\quad\quad idiff \leftarrow (II + 1) \cdot \left\lfloor \frac{irange}{stepsize} \right\rfloor - II \cdot \left\lfloor \frac{irange}{stepsize} \right\rfloor$
24: $\quad\quad jdiff \leftarrow (JJ + 1) \cdot \left\lfloor \frac{jrange}{stepsize} \right\rfloor - JJ \cdot \left\lfloor \frac{jrange}{stepsize} \right\rfloor$
25: $\quad\quad imicroTemplate \leftarrow microcellI_{idiff,jdiff}$
26: $\quad\quad jmicroTemplate \leftarrow microcellJ_{idiff,jdiff}$
27: $\quad\quad$ **repeat**
28: $\quad\quad\quad$ *process loop body*(i, j)
29: $\quad\quad\quad i \leftarrow II + (imicroTemplate$ $\textbf{and}_{\text{bitw}}$ $7) - 5$
30: $\quad\quad\quad j \leftarrow JJ + (jmicroTemplate$ $\textbf{and}_{\text{bitw}}$ $7) - 5$
31: $\quad\quad\quad imicroTemplate \gg 3,\ jmicroTemplate \gg 3$
32: $\quad\quad$ **until** $imicroTemplate = 0$
33: $\quad\quad Z \leftarrow Z + 1$
34: $\quad$ **end while**
35: **end for**

macro can be used in any host algorithm with any arbitrary minimum and maximum boundaries and keeps the algorithm well structured. Additionally, this has significant performance benefits. The whole algorithm can be implemented as a single method. The compiler has all options for optimizations, including extraction of loop invariants, and loop unrolling can be made fully automatic by the compiler.

We also tested an implementation based on an object-oriented approach because there, we could use a strategy pattern to eliminate the branching with the `if`-statement in line 14-18 (Algorithm 6). The strategy pattern could determine beforehand whether we have an i-dominant rectangle or an j-dominant rectangle. However, it turned out that this abstraction induced an overhead that was larger than its performance gain. We believe that this is due to optimizations based on the compiler, which could be applied in the macro version but not in the object-oriented version.

8.3 Parallelization

We combine our Morton-order curve with explicit parallelization techniques to enhance the efficiency of our algorithm. On the coarse-grained level, Parallel threads for different cores of the CPU can be managed by concepts like OpenMP or CILK. We are using OpenMP for the Multiple Instruction Multiple Data (MIMD) parallelization, where each of the processors works on an independent part of our matrix. The parallelization using the construct "#pragma omp parallel" is packed into the pseudo-code keyword **MORTON-ORDER**, where each core works on independent chunks of data. The typical usage pattern of our Morton-order loop is outlined in Listing 8.1.

At line 6, we determine which thread is using which part of the matrix. We are using the Morton-order curve to loop over cells with a size of 24×8. These relatively large cells enable us to utilize the full bandwidth of AVX-512. AVX-512 enables Single Instruction Multiple Data (SIMD) parallelism, which is commonly known as vectorization.

AVX-512 intrinsic functions use data types with a length of 512 bits (=8 double precision) as operands representing the registers in Intel's Xeon Phi and Skylake-X CPUs. Each computational core is equipped with 32 registers, and the matrix multiplication or LU decomposition, which are clearly I/O bounded problems. In our source code, we load 3 consecutive 512-bit registers ($3 \cdot 8 = 24$) and multiply them with a value of each of the 8 rows. This means in each iteration we take a cell of A build a vector using *_mm512_set_pd* and multiply it with a vector of B and store the scalar product from

Listing 8.1: Morton-order usage pattern

```
#pragma omp parallel
{
  ...
  ME=omp_get_thread_num();
  T=omp_get_num_threads();
  imin=ME*T*n/24;imax=(ME+1)*T*n/24;
  jmin=0;jmax=m/8;
  ZORDER_START(i,j,imin,imax,jmin,jmax){
    ...
    for ( ... ){ ...
    }
    ...
  }ZORDER_END(i,j)
}
```

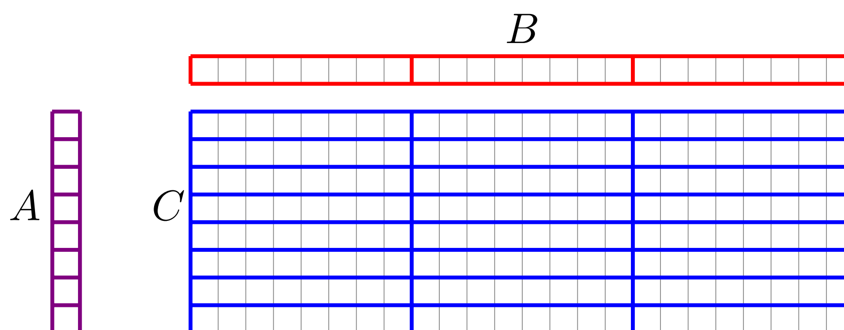


Figure 8.8: SIMD parallelization.

_mm512_fmadd_pd in the corresponding C vector (c.f. Figure 8.8). This technique enables us to utilize the full bandwidth of AVX-512.

8.4 Application of Morton-order Loops for LU Decompositon and Matrix Multiplication

There are many different variants for dense LU factorization. A set of n linear equations in n variables is solved by performing LU factorization and solving the resulting triangular systems. There are many variants like Crout's algorithm [107], which performs an in-place factorization, but it is typical to transform this problem to a block LU decomposition. The level of granularity for this block factorization range from a simple case of four blocks,

which will be outlined here, to a large case of panel factorization [59, 80] or constructed dependency graphs using directed acyclic graphs [84]. The major benefit of applying a block algorithm here is to exploit the properties of the matrix multiplication. The matrix multiplication is I/O bounded, and therefore it enables better performance for the LU decomposition. We consider the simple case of a block LU-factorization $A = LU \in \mathbb{R}^{n \times n}$, where the matrix $A \in \mathbb{R}^{n \times n}$ is nonsingular. Then we can write:

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \cdot \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix} \tag{8.2}$$

Algorithm 7 LU block algorithm

Input: $A \in \mathbb{R}^{n \times n}$

Output: $L, U \in \mathbb{R}^{n \times n}$

1: Compute L_{00} and U_{00} by factorizing $A_{00} = L_{00}U_{00}$
2: Compute L_{10} by solving $L_{10} = U_{00}^{-1}A_{10}$ using backward substitution
3: Compute U_{01} by solving $U_{01} = L_{00}^{-1}A_{01}$ using forward substitution
4: Compute the Schur complement of $L_{00}U_{00}$ which is $A_{11} = A_{11} - L_{10}U_{01}$ using parallel matrix multiplication
5: Compute the block LU factorization for $L_{11}U_{11}$ recursively on A_{11}

The block LU factorization is outlined in Algorithm 7. The code is implemented as an in-place version, but we present the pseudo-code as an out-of-place variant for the sake of better readability. In the first step (line number 1) we perform a LU decomposition on a small block using a straight forward parallel implementation of a row block LU factorization [86]. Unfortunately, suppose we apply a space-filling curve in this step. The effect will be very little because we need at least three loops (two loops to describe the traversal of the space-filling curve and one additional inner nested loop) to apply our technique successfully. There are only two for the elimination step. The next steps in the block LU factorization algorithm are the backward substitution, the forward substitution and the matrix multiplication (lines 2 to 4 in Algorithm 7). Each of these algorithms will be explained individually in the following.

Algorithm 8 Canonical LU decomposition

Input: matrices $A_{00} \in \mathbb{R}^{n \times n}$
Output: $L_{00} \in \mathbb{R}^{n \times n}$ and $U_{00}^{-1} \in \mathbb{R}^{n \times n}$
1: **for** $k \leftarrow 0$ to n **do**
2: **for** $i \leftarrow k + 1$ to $i < n$ **do**
3: $a_{i,k} = a_{i,k}/a_{k,k}$
4: **end for**
5: **for** $i \leftarrow k + 1$ to $i < n$ **do**
6: **for** $j \leftarrow k + 1$ to $j < n$ **do**
7: $a_{i,j} = a_{i,j} * a_{k,j}$
8: **end for**
9: **end for**
10: **end for**

The canonical LU decomposition in Algorithm 8 is based on a row-cyclic algorithm for LU decomposition [86]. This canonical LU decomposition is the first step in our block algorithm, where we compute L_{00} and U_{00}, by factorizing $A_{00} = L_{00}U_{00}$. This part is an example where the Morton-order curve has minimal impact on performance and cache-efficiency. In this algorithm all computations for $k = 1$ depend on the previous computations for $k = 0$. Therefore, the parallelization is applied on both inner for loops at line 2 and 5. There are two-nested loops left, but we would need three nested loops to apply our technique successfully. The two outer loops shape the Morton-order curve, and the additional inner loop supports the Morton-order in addressing the L1 cache. As we will see in the experiments section, the Morton-order curve optimizes L2 and L3 cache access. However, since this part of the LU decomposition is often very small, this has minimal impact on the performance and certainly not on larger cache hierarchies than the L1 cache.

Algorithm 9 Backward substitution algorithm

Input: matrices $U_{00} \in \mathbb{R}^{n \times n}$ and $A_{10} \in \mathbb{R}^{m \times n}$
Output: $L_{10} \in \mathbb{R}^{m \times n} = U_{00}^{-1} A_{10}$
1: **MORTON-ORDER** $(l, k) \in \{0, \ldots, m - 1\} \times \{0, \ldots, n - 1\}$ **do**
2: **for** $i \leftarrow 0$ to $i < k$ **do**
3: $l_{l,k} = a_{l,k} - (a_{l,i} \cdot u_{i,k})$
4: **end for**
5: $l_{l,k} = l_{l,k}/u_{k,k}$
6: **END-MORTON-ORDER**

In the next step, we apply backward substitution to obtain L_{10}. We are using

OpenMP for the parallelization step, where we partition A_{10} among several threads. Each of the threads updates an equal number of columns from A_{10} and computes the backward substitution for its part of A_{10}. The algorithm is outlined in Algorithm 9, where we loop over the A_{10} matrix using our Morton-order curve. Our curve traverses over single cells of the matrix, but parts of this matrix are unrolled by an optimized loop unrolling strategy. In the case of the backward- and forward-substitution, this is summarized parts of 8 by 8 matrix cells.

The forward substitution algorithm works similarly, and it is outlined in Algorithm 10, where we partition the matrix A_{01} among the threads and compute the resulting part of the U_{01} matrix independently. Both substitution algorithms perform on a triangular matrix, and in order to apply a space-filling curve, we need to ensure that the curve fulfills the monotonicity properties because of the data dependencies to the left and the top.

Algorithm 10 Forward substitution algorithm

Input: matrices $L_{00} \in \mathbb{R}^{n \times n}$ and $A_{01} \in \mathbb{R}^{n \times m}$
Output: $U_{01} \in \mathbb{R}^{n \times m} = L_{00}^{-1} A_{01}$
1: **MORTON-ORDER** $(k, i) \in \{0, \ldots, n-1\} \times \{0, \ldots, k-1\}$ **do**
2:     **for** $j \leftarrow 0$ to m **do**
3:         $u_{k,j} = a_{k,j} - (a_{i,j} \cdot l_{k,i})$
4:     **end for**
5: **END-MORTON-ORDER**

Matrix multiplication is an algorithm where any space-filling curve can be applied straightforwardly because it has no general data dependencies. For the forward and backward algorithm, the space-filling curve for the matrix multiplication traverse through groups of matrix cells, which are summarized by loop unrolling strategy. For the matrix multiplication, we are using 8-by-24 cells. The basic algorithm from our introduction is outlined in Algorithm 11. It essentially multiplies the rows of $A \in \mathbb{R}^{m \times p}$ with the columns of $B \in \mathbb{R}^{p \times n}$. If we consider large matrices, where the rows of $a_{\cdot,k}$ and $b_{\cdot,k}$ are too large for the small caches, then we need to decompose the matrices horizontally into groups of s (Algorithm 11 line 3 to 5). We use OpenMP for the parallelization, where we partition an imaginary grid of the matrix C among the threads so that the rows from matrix A and B are distributed equally.

Algorithm 11 Matrix multiplication

Input: matrices $A \in \mathbb{R}^{m \times p}$, $B \in \mathbb{R}^{p \times n}$ and $C \in \mathbb{R}^{m \times n}$
Output: $C = AB - C$

1: **for** $K \leftarrow 0$ **to** $p-1$ **by** stepsize s **do**
2: **MORTON-ORDER** $(i,j) \in \{0,\dots,m-1\} \times \{0,\dots,n-1\}$ **do**
3: **for** $k \leftarrow K$ **to** $\min\{K+s,p\}-1$ **do**
4: $c_{i,j} = c_{i,j} - (a_{i,k} \cdot b^T_{j,k})$
5: **end for**
6: **END-MORTON-ORDER**
7: **end for**

8.5 Experimental Evaluation

We share our code to make our experiments transparent and comprehensible. Code and experimental data is available[1].

Microarchitecture and Evaluation Measures

Our experiments have been performed on Intel© Xeon Phi™ 7210 codename Knights Landing (KNL) with 1.3 GHz and 64 cores, 96 GB main memory, and CentOS 7.4.1708 as the operating system. A KNL processor socket has 32 active tiles, where each tile consists of two cores. Each core has 32 kB instruction and 32 kB data cache for L1 and supports AVX-512 SIMD instructions. The L2 cache of 1 MB is shared among two cores within a tile. The KNL processor family has an improved cache and memory organization, where one can choose between five different clustering modes (see `https://colfaxresearch.com/knl-numa/` and three different memory modes (see `https://colfaxresearch.com/knl-mcdram/`). For all our experiments, we have used the quadrant mode in combination with the cache-mode since this configuration complies with multi-core commodity hardware[67]. Our configuration results in a 16 MB shared L3 cache among all cores. We investigated the cache access pattern on four different levels, namely L1, L2, L3 clean (i.e., L3c), and L3, allowing dirty reads (i.e., L3d). We describe the cache pattern with a metric defined as *cache miss-rate*, which is defined for each individual cache level as *1-cache hitrate* [67]. Cache-oblivious algorithms should perform well on different hardware infrastructure, so we decided to evaluate our approach to a second different CPU infrastructure, our Xeon. The Xeon (Skylake) server is equipped with 16 cores, each with a base frequency of 2.1 GHz, but the frequency can increase up to 3.7

[1] `https://gitlab.cs.univie.ac.at/martinp16cs/mortonlu`

GHz under heavy load. Each of the cores is equipped with AVX-512 SIMD instructions, 32 kB L1 data cache, and 1 MB L2 cache. The L3 cache with 22 MB is shared among all cores. The operating system is Ubuntu Linux 18.04.4 LTS. All experiments have been performed with 50 repetitions and the GNU compiler version 9.1.0.

Arbitrary Sized Morton-order

We evaluate our Morton-order variants against five different comparison methods. We start with the problem of applying the Morton-order to arbitrarily sized matrices. This problem is usually addressed with padding [34, 122, 130], but we show the performance advantage of microcells in contrast to such approaches. This implementation does not store any intermediate results and is labeled as “Dilated (padding)”. In this approach, we proceed as follows. We calculate the largest power of 2, which just fits into the matrix. Algorithm 4 is then applied until this sub-matrix is completely processed. If this sub-matrix fits again into the initial matrix, we repeat the algorithm one more time until we have to reduce the sub-matrix to the next smaller power of 2. For our running example in Figure 8.7, which is a quite uncommon scenario, we fill the 13×7 grid with 3 matrices of order 2^2, 6 matrices of order 2^1 and 19 single cells.

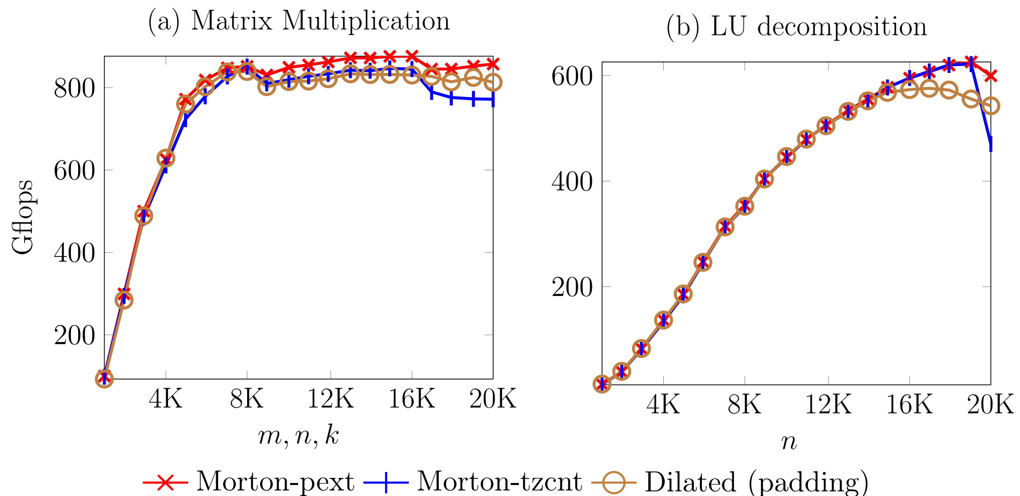


Figure 8.9: Comparison of different Morton-order generation approaches (Xeon-Phi).

In our Morton-pext approach we combine our microcells approach together with the pattern extraction (pext) as described in Algorithm 6. The second approach, called Morton-tzcnt is a combination of our microcells with a Morton-order generated by the trailing zeros approach (tzcnt) described in Algorithm 5.

In Figure 8.9 we compare these two approaches for the matrix multiplication and LU decomposition. In the case of the matrix multiplication in the Figure on the left, the main thing to notice here is if we generate a large curve, the dilated approach's performance and the performance of Morton-tzcnt drops down. The performance of Morton-pext is stable. The runtime for the dilated approach using padding is 19.80, and the runtime for Morton-pext is about 18.78 for a matrix size of 20 000. These runtimes show a performance gain of 5.4% for microcell support. Our implementation of Morton-tzcnt has a runtime of 20.96 seconds.

For the LU decomposition, we observe a similar performance behavior. The runtime for Morton-pext, which is the fastest method has a runtime of 8.96 seconds. The runtime for the dilated approach is about 10.88 seconds and for Morton-tzcnt the runtime performance drops down from 621 GFlops and 7.44 seconds to 470 GFlops and 11.43 seconds.

The results pesented here in the Figure 8.9 are difficult to read, so we have again attached the runtimes in tabular form in the appendix, c.f., Table A.1 for matrix multiplication and Table A.2 for the LU decomposition.

Morton-order Matrix Multiplication

The runtime performance for the method Morton-tzcnt was poor compared to the other approaches. For the remaining part of this chapter, we focus on our method Morton-pext, which has superior performance properties. In the following experimental evaluation, Morton-pext is the baseline for both Morton-order curves, the Z-order and *И*-order.

In the following, we compare our approach against the current state-of-the-art algorithms and evaluate the gain of using microcells by testing the performance against a method that traverses in Morton order but having microcells not implemented. We label this method with "plain M", for plain Morton. We look-ahead and calculate the next power-of-2 i and j values and store them intermediately with this method. The canonical order is an interesting baseline since, for C-like languages, we have a row-oriented memory layout. Bader et al have introduced the Peano curve. as a recursive storage scheme for matrix elements in cache-oblivious algorithms [9, 12] and is implemented in a framework called TifaMMy[2]. The Hilbert curve has been introduced by Böhm et al. and is implemented exactly as our approach except for the Hilbert curve [24] traversal

[2]obtained by `https://sourceforge.net/projects/tifammy/`

instead of using Morton order.

A textbook example for an application of cache-oblivious algorithms is matrix multiplication. Since we do not have a data dependency here, we can apply all different comparison methods. A clear picture emerges from the evaluation of the performance of all different methods depicted in Figure 8.10a. Three methods show a more or less similar performance behavior, namely Z-order, Hilbert, and И-order. A closer look at these methods for matrices with a size of $m = n = k = 20\,000$ that the И-order is superior with 875 GFlops, followed by Hilbert with 840 GFlops and the Z-order with 810 Gflops. These three methods are approximately twice as fast as the Peano curve with 395 Gflops or the canonical variant of our approach with 344 Gflops. The cores for Xeon Phi are relatively slow (1.3 GHz), and the plain Morton curve generates the upcoming curve pattern for the next power of 2 values. This leads to some values of n to an inefficient computation. The average runtime of plain Morton is roughly 5.15 seconds for a matrix size of 12 000, and 3.93 seconds for the И-order . This is a performance improvement of more than 30%.

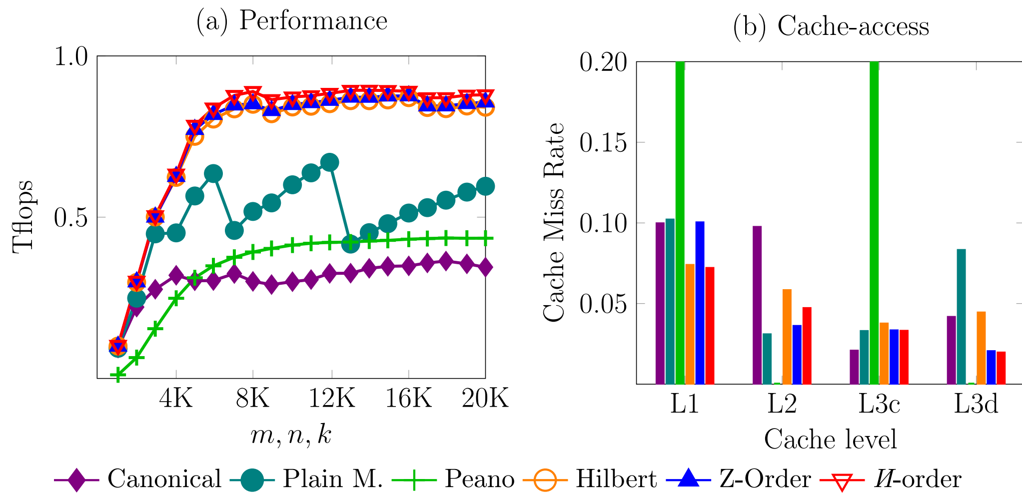


Figure 8.10: Matrix-multiplication on Xeon-Phi.

The effects of the space-filling curves on the cache access pattern is outlined in Figure 8.10b. Considering the cache as a whole, the И-order has a relatively low miss-rate, followed by Hilbert and the Z-order. For the Xeon-Phi processor, there are two cores together on a “tile”, which share the same L2 cache, and the L3 cache is shared among all cores. Therefore, the L1 and L2 cache, as well as the L3c and L3d cache, should be considered together. There is a high cache miss-rate (more than 60% for L1 and L3c) for

the Peano approach, but almost all (>99%) of the access in L2 and L3d hit. The largest difference between the Morton order approaches and the canonical approaches is in the L2 cache. We discuss this phenomenon in Section 8.6.

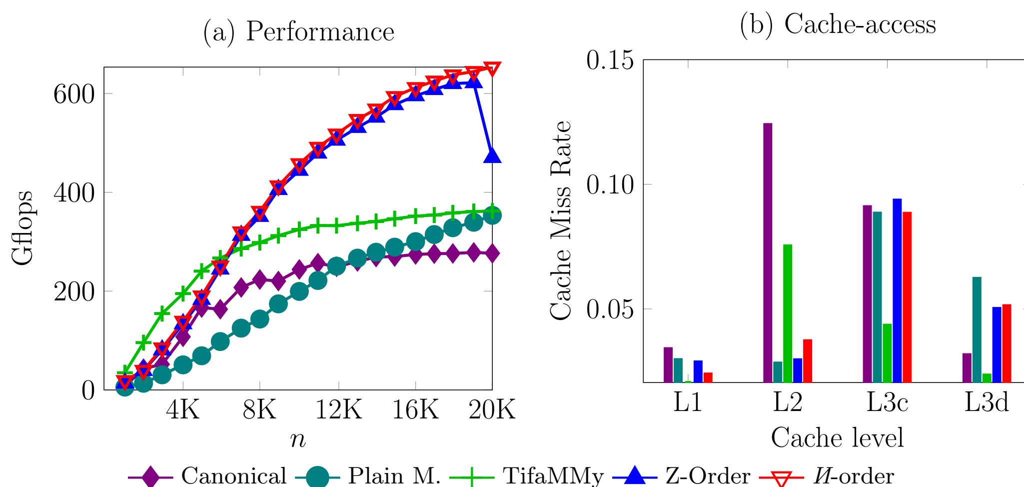


Figure 8.11: LU decomposition on Xeon-Phi.

LU Decomposition

For the LU decomposition (cf. Figure 8.11a) we have a similar picture, as seen before for the matrix multiplication. Both Morton order methods are superior compared to the canonical order and the block recursive scheme for LU decompostion (TifaMMy). For a matrix size of $n = 20\,000$. The *И*-order has a performance of 653 GFlops and the Z-order curve performance of 612 Gflops, whereas TifaMMy is around 362 Gflops and the canonical approach lags with 276 Gflops. The gain in runtime performance of the *И*-order in comparison to the plain Morton curve is roughly 80% for a matrix size of $n = 20\,000$.

We captured the cache-access pattern and represent it in Figure 8.11b. Considering the L1 and L2 cache as a joint access pattern, both Morton methods show a more efficient cache-access in L1 and L2 than the canonical order and TifaMMy. The cache-access pattern is also closely coupled with the runtime performance. For slow algorithms, it tends to have a better cache-access pattern since the method has more time to access the cache (cf. TifaMMy for L3c and L3d). For the LU decomposition, we also measured the time spent in forward and backward substitution illustrated in Figure 8.12. For both substitution algorithms, we have similar behavior in the runtime. The *И*-order is followed

closely by the Z-order curve, and the canonical variant lags behind. The gain of *И*-order compared to plain Morton is for the forward substitution roughly 25% and the backward substitution roughly 41% for a matrix size of $n = 20\,000$.

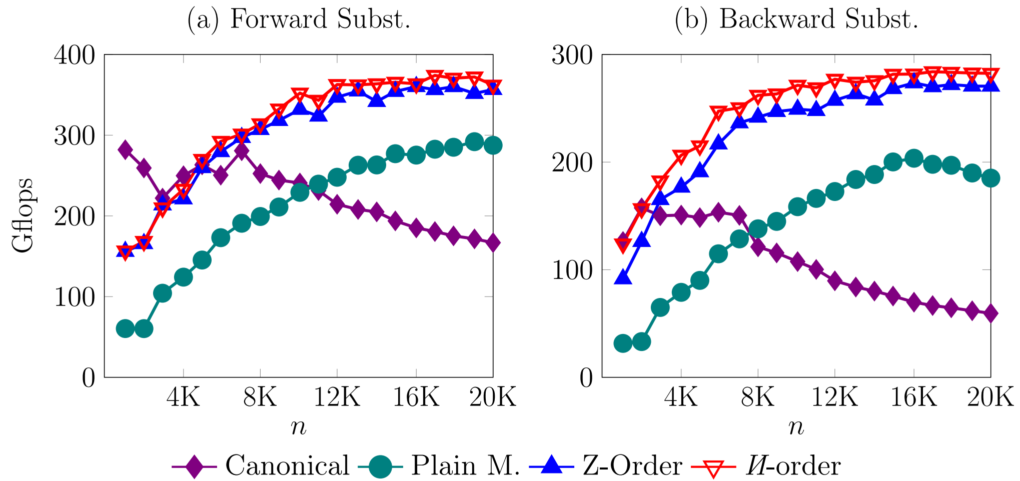


Figure 8.12: Forward and backward substitution.

Performance on Different Hardware

Cache-oblivious algorithms tend to perform well on different hardware systems. The results evaluated on our Xeon server are depicted in Figure 8.13. In Figure 8.13a we have evaluated the matrix multiplication and in Figure 8.13b the LU decomposition algorithm. We see a similar runtime behaviour compared to Figure 8.10a. However, the difference between the methods is less evident. For the matrix size of $N = 20\,000$, we observe a superior performance for *И*-order with 644 Gflops, and both approaches Z-order and Hilbert perform more or less the same with 636 Gflops for Z-order and 635 Gflops Hilbert. The canonical approach is around 297 Gflops and TifaMMy around 285 Gflops. In the case of the matrix multiplication, the advantage of *И*-order (644 Gflops) over our plain variant (607 Gflops) is about 6% and for the LU decomposition roughly 11%.

The difference in runtime between the *И*-order and the Z-order curve is somewhat more visible for the performance of the LU decomposition in Figure 8.13b, since the LU decomposition includes the forward and backward algorithm, as well as the matrix multiplication. For the matrix multiplication and a matrix size of $m, n, k = 20\,000$ in Figure 8.13a the *И*-order has 644 Gflops, followed by 635 Gflops Hilbert and 636 Gflops

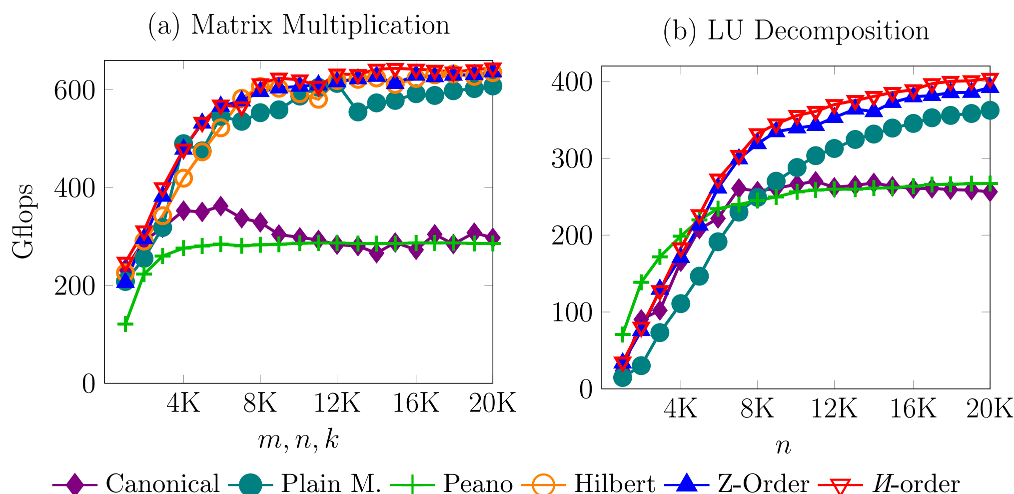


Figure 8.13: Results evaluated on Xeon.

for Morton. These methods' performance pattern is on both servers very similar if we compare it to the LU decomposition on the Xeon Phi in Figure 8.11. The И-order leads with 403 Gflops, followed by the Z-order curve as a runner-up at 392 Gflops. TifaMMy and the canonical variant share third place at 267 Gflops and 256 Gflops.

8.6 Related Work and Discussion

We introduced two different methods, Morton-pext and Morton-tzcnt. We combined both methods with our microcells and in the experiments, we have seen a superior performance for Morton-pext. We believe that the performance drop is because the branch prediction by the compiler for the `if` in Algorithm 5 at line 6 fails to predict the correct branch. Based on Morton-pext, we have two different Morton-order traversals based on the *Z*-order and on the *И*-order. Both traversals show superior performance in comparison to the state-of-the-art in the literature.

Cache-oblivious Algortihms

The notion of *cache-obliviousness* has been first introduced by Frigo et al. in 1999 [55]. A cache-oblivious algorithm performs well on any multi-level memory hierarchy without knowing the structure and the hierarchy parameters, e.g., cache and memory size, transfer block size, and bandwidth. The two fundamental design patterns of cache-oblivious algorithms are localized memory access pattern and a divide and conquer approach.

Space-filling curves integrate both ideas. Morton presented his ordering in 1966 to index frames in a geodetic database [89]. It has been applied to several algorithms and data structures for tasks like sorting, searching, or indexing [100, 130] as well as communication strategy among nodes [75] in distributed systems.

Most related to our work, Bader et al. proposed to use the Peano curve for matrix-matrix multiplication [9, 12] and LU decomposition [62]. Both algorithms are combined in one framework, which is called TifaMMy. The Peano approach's implementation is a recursive block scheme, whereas our approach is based on non-recursive Morton order loops. If the blocks in such a recursive scheme do not fit into the cache, it leads to such high miss rates. We have tested various block-sizes without success, and we, therefore, infer that these are effects of optimized building blocks. The Hilbert-curve [24], is slower then the *И*-order and cannot be applied to LU decomposition because the Hilbert curve does not fulfill the monotonicity properties. Compared to our canonical approach, both algorithms' nested loop structure, the matrix multiplication, and the LU decomposition are reflected in the cache-pattern. Our space-filling curves target the L2 cache to a greater extent. Previous approaches on Morton-order curves have been applied in Single-Threaded environments, [123, 130] but they are limited to $n \times n$ arrays, where n is to the power of 2 or 3.

Morton order storage's common drawback is that straightforward application is only possible for square matrices whose side is an integer of power of two. Other matrix sizes require special treatment, such as padding [33, 34, 130]. Our implementation called "Dilated padding" is directly comparable with the approach proposed by Valsam and Skjellum [122]. In this approach, the authors propose to use small kernel operations on the matrix multiplication and traverse these kernels in a Morton-order by recursively divide the matrix into 2 by 2 submatrices of where one part is as large as the greatest power of 2. In some cases, this leads to large jumps, where the data locality is not preserved. As an example, think of a 5×5 matrix. After the 2×2 part is finished, there is a reset to the matrix's minimum boundaries. Such large resets do not occur in microcell approaches. Therefore it is more data-local.

Applications of Space-filling curves

Morton-order curves are vastly used in various research areas. The basic idea is to sort the input data set according to the Morton-order. Once sorted, the points can be stored in a binary search tree, or more specifically in a linear [56] or pointer-based quadtree

[109]. In magnetic resonance imaging (MRI) [115] space-filling curves reduce the readout time and increase the peak signal-to-noise ratio (PSNR). In computer graphics rendering a 3D Morton-order is used to build a high-resolution 3D voxel grid for efficient I/O search [13]. The authors developed a library, called "libmorton" which allows the user to encode $m = morton\,(i, j)$ and decode $(i, j) = morton^{-1}(m)$ Morton-order values. Unfortunately, this library is restricted to squarred grid sizes. Similar to sorting, there exists some transformation techniques on tensors, where the tensors are reordered according to a blocked Morton ordering [82, 100].

Space-filling curves, such as the Morton-order, have also been applied to hardware systems different than the CPU, such as distributed systems [75] and GPUs [93, 121]. The Morton order reduces the latency for distributed systems based on idealized communication patterns [75]. In the case of the GPU, the memory hierarchy consisting of global, local, and texture memory which can be efficiently walked through [93] or indexed using the Morton-order curve [44, 79].

Optimized Techniques for Specific Tasks or Hardware

The Math Kernel Library (MKL) is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS (Basic Linear Algebra Subprograms) [47] and LAPACK (Linear Algebra PACKage) [6]. BLAS is a de facto standard for low-level routines in linear algebra, such as the matrix multiplication routine DGEMM (Double precision GEneral rectangular Matrix- matrix Multiply). LAPACK is a standard software library for numerical linear algebra, such as matrix factorizations like the LU decomposition, such as the routine DGETRF (Double precision GEneral TRiangular Factorization), as well as forward and backward substitution routines. Intel MKL library achieves similar performance as OpenBLAS [138], and the implementation of OpenBLAS is heavily tailored with many hand-crafted optimizations for specific processor types and therefore not a suited comparison method especially for cache-oblivious algorithms. Furthermore, such approaches transform the layout of matrices to optimize I/O efficiency. State-of-the-art approaches for distributed computing are libraries such as ScaLAPACK [38] or BLACS `https://software.intel.com/en-us/mkl-developer-reference-c-blacs-routines`. Special cases, such as sparse LU decomposition on hybrid systems [136] have been published as well. Other algorithms, such as [59] build upon the ATLAS framework by using low-overhead kernel primitives generated by the ATLAS framework. Furthermore, they suggest profiling critical path operations, where any element should be prototyped and timed. Such critical paths of

existing highly tuned building blocks could be tracked in a directed acyclic graphs [84].

These optimized variants are far beyond our research scope since we want to demonstrate the positive effects of Morton-order traversal. The LU decomposition is a suitable application since it has requirements on monotonicity properties. The performance of such depends on the matrix-multiplication, where we can compare to other space-filling curves.

8.7 Conclusion

In this chapter, we introduced the Morton order curve, a space-filling curve that enables us to exploit the properties of today's modern cache hierarchies. There are two different variants, the Z-order curve and the *И*-order with slight performance advantages for the *И*-order. Here, we have applied our Morton-order curves to Multi- and Many-core environments. Our microcells concept allows us to fill any arbitrary $n_i \times n_j$ rectangle and gives performance advantages of up to 80%, depending on the application and the underlying hardware. The Morton-order curve is not restricted to data dependencies in comparison to Peano or Hilbert. These properties make our Morton-order curve particularly attractive to replace pairs of nested loops in host algorithms. We believe that many algorithms in the field of linear algebra would profit from a Morton order traversal.

Chapter 9

Energy efficiency on Data Movement

Microprocessor performance scaling in recent years has been achieved by scaling throughput, i.e., by processing more threads concurrently by an increasing number of cores and employing techniques, such as Simultaneous MultiThreading (SMT) and vectorization (i.e., SIMD). While there is an increasing demand for computational power, the number of processing cores and threads are limited by the power and energy budget. A large consumer of this energy budget is the memory hierarchy. How much energy of this budget is occupied depends on the system used and on its application. Modern microprocessors with a rich memory hierarchy usually consume around 50% [124], whereas programmable processors consume in average around 70% [42]. An extreme case is video encoders where the memory to power consumption is up to 90%, including off-chip memory communication [119].

The space-filling curves proposed in this thesis have different traversals for matrices and different data access patterns. Therefore, we think another interesting aspect of our space-filling curves is to look at energy consumption and efficiency. In this chapter, we compare different space-filling curves for their energy behavior and measure their cache-access patterns to draw sound conclusions. We start with an introduction in section 9.1, followed by the experimental setup in section 9.2. Our experiments are described in section 9.3. We discuss our results in section 9.4 and conclude in section 9.5.

9.1 Introduction

In the early days of energy-efficient computing, there have been some estimates of application-specific integrated circuits (ASIC). These are programmable processors cus-

tomized for a particular use rather than integrated for general-purpose use like microprocessors. For example, a chip designed to run in a digital voice recorder or a high-efficiency Bitcoin miner is an ASIC. For such chips, the main responsibilities for the cache are supplying data and instructions. These consume 70% of the processors energy, divided as 28% for data and 42% for instructions supply [42]. Performing arithmetic consumes 6%, and performing clock- and control-logic consumes 24%. After all, microprocessors from commodity hardware or in the server segment often have a higher clock frequency and have far more computing cores and a richer cache hierarchy. Actual numbers are difficult to determine as the cache hierarchy's power consumption cannot be measured independently. Educated guesses for the energy consumption of the memory hierarchy in the literature range are around 50% [124]. Jenga, a system that eliminates access to unwanted cache levels to improve performance and energy-efficiency, saves 23% energy on average [120].

9.2 Experimental Setup

Measuring Power with a Watt Meter

Using a watt meter is the easiest and most accurate approach to measure power and energy efficiency. There are also other APIs that allow reading internal performance counters, such as Running Average Power Limit (RAPL) [127], but these were not available for Xeon Phi x200 (only for coprocessors, offloaded from a host system). The power meter in use was Hioki 3334 AC/DC Power HiTESTER with a measurement accuracy of ±0.1%. The 3334 is an AC/DC power meter that measures inrush current and power consumption, ideal for DC, and current and power integration applications to meet energy efficiency standards. Ideally, a programmer wants to know the power measurements associated with specific parts of their application. Such information allows the program to be modified to both improve performance and be more power-efficient. For this reason, we developed a C/C++ interface to query the current state and perform energy measurements during our program's runtime. Our machine of interest is connected to the power meter over the RS-232C interface using a crossover cable (see Figure 9.1).

Furthermore, we have developed a program to start, stop, reset or query current power consumption from our watt meter. The code is available to the public[1].

[1] `https://gitlab.cs.univie.ac.at/coloops/HIOKI-3334`

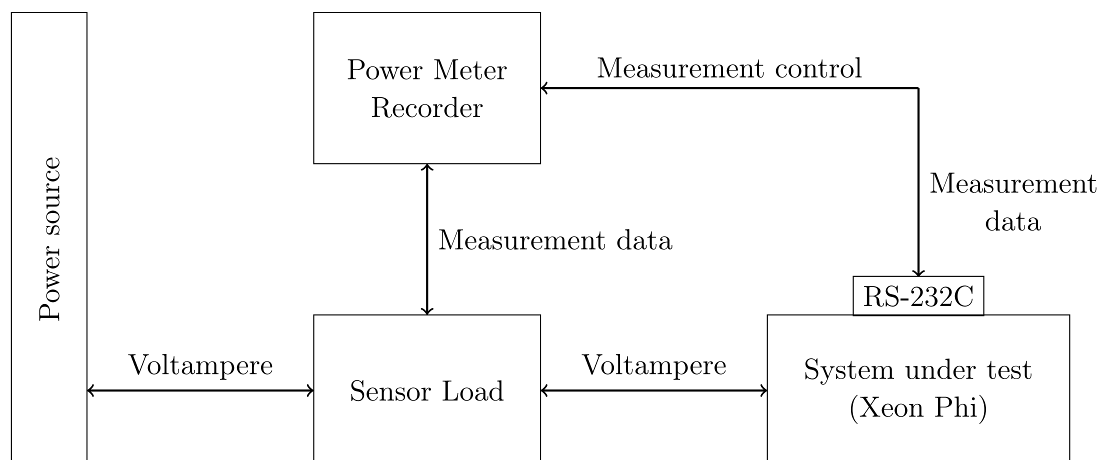


Figure 9.1: Power meter set-up.

Evaluation Metrics

The performance is reported as the number of floating-point operations executed per second (FLOPS/s), while the energy-efficiency corresponds to the number of operations per Joule (OPS/J). We consider the number of operations, GFLOPS, for the matrix multiplication, which is defined as $10^9 \cdot 2 \cdot mnk$, where m, n and k define the size of the matrices which are multiplied. The matrix multiplication has a runtime complexity of $\mathcal{O}(nmk)$, and in the innermost loop, we have two operations, one multiplication and one addition. For the LU decomposition where we factorize A into $A = LU$, and where A is nonsingular of an order of n, we have $\frac{2}{3}n^3$ floating-point multiplications and additions. This approximation is commonly used to estimate the number of operations [131]. The watt meter measured energy consumption, which indicates the measures in Watt-hours [Wh]. This specific measure represents a constant power over a period of time. A related measure, commonly used in the literature is Watt seconds [Ws]. One Watt second is equivalent to 1 Joule [J]. Our energy efficiency is expressed in a ratio of Million Operations per Second (MOPS) divided by Wh [MOPS/Wh]. The conversion to [OPS/s] or [OPS/J] can be easily done with one multiplication by a factor of 3600.

9.3 Experimental Evaluation

We compare different approaches for the matrix traversals in terms of energy efficiency, based on better data movement. Therefore, we need to set the runtime performance in relation to energy efficiency. The performance evaluation in terms of 10^9 (=Giga)

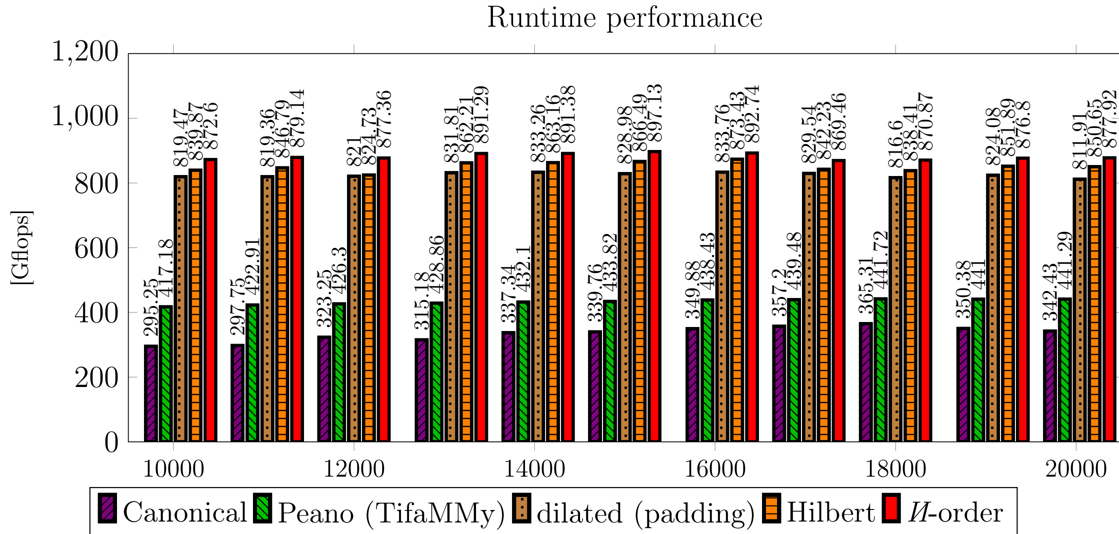


Figure 9.2: Runtime performance of the matrix-multiplication on Xeon-Phi.

floating-point operations [Gflops] is presented in Figure 9.2. The canonical order serves here as an interesting baseline, where each consecutive row is traversed from the beginning to the end. The matrix multiplication that follows a recursive storage scheme in an order defined by a Peano curve for matrix elements has been introduced by Bader et al. [9, 12] and is implemented in a framework called TifaMMy[2]. The Dilated (padding) approach follows a Morton-order, where the matrix is padded with sub-matrices, each with side lengths equal to a power of 2. The Hilbert-order has been introduced in chapter 5 as well as the *И*-order in chapter 8.

For the matrix multiplication a clear winner in terms of runtime performance, energy consumption (c.f. Figure 9.3) and energy efficiency (c.f. Figure 9.4) is the *И*-order. In all three categories, the Hilbert curve is the runner up, followed by the dilated (padding) approach. All of these named approaches are at least twice as fast as the canonical traversal of the matrix.

For the LU decomposition there is a very similar pattern in terms of runtime performance (c.f. Figure 9.5), power consumption (Figure 9.6 and energy efficiency (Figure 9.7). The Hilbert curve is left out in this comparison, because of its data dependencies issues (see chapter 8).

[2]obtained by `https://sourceforge.net/projects/tifammy/`

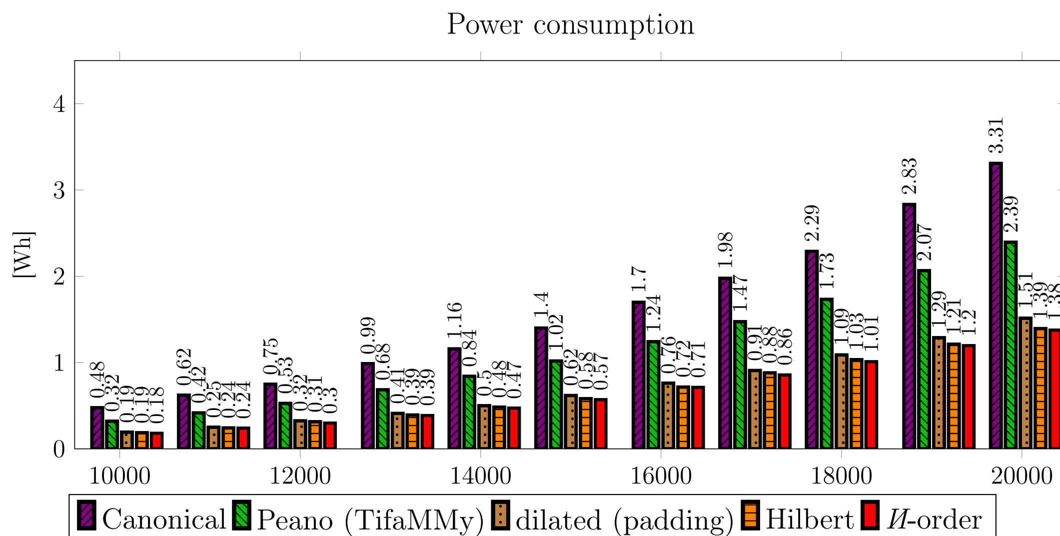


Figure 9.3: Power consumption of the matrix-multiplication on Xeon-Phi.

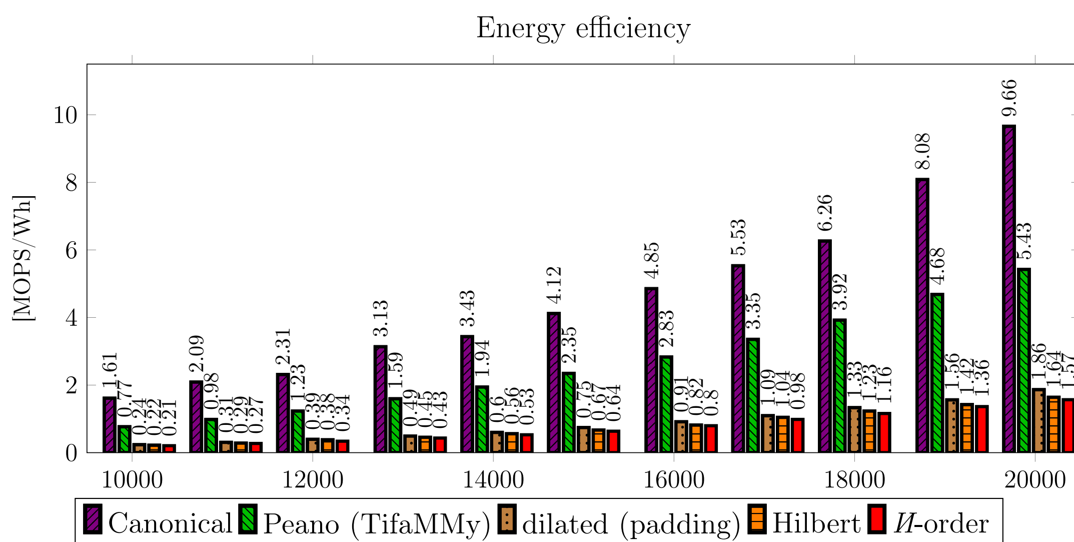


Figure 9.4: Energy efficiency of the Matrix-multiplication on Xeon-Phi.

9.4 Discussion

The overall impression would lead to a conclusion, where the И-order traversal performs best in cases of runtime performance, has the lowest energy consumption, and is, therefore, the most energy-efficient solution. However, the hidden variable here is the runtime, which influences both the energy consumption and the runtime performance in terms of

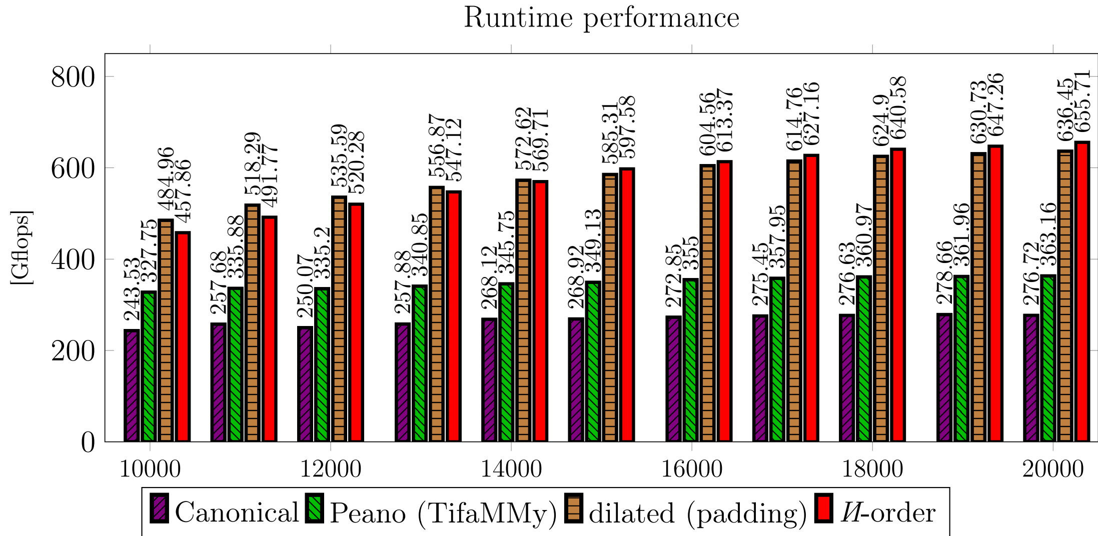


Figure 9.5: Runtime performance of the LU decomposition on Xeon-Phi.

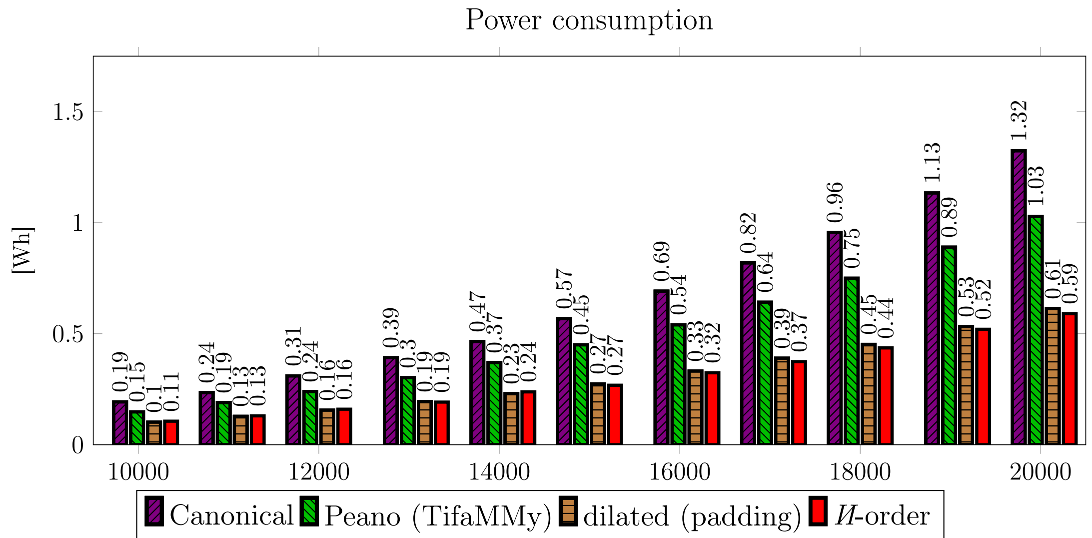


Figure 9.6: Power consumption of the LU decomposition on Xeon-Phi.

Gflops. Therefore, it is more an indicator of being energy efficient and not proof. Another indicator is the cache access pattern. The *И*-order has a lower number of expensive cache misses (L2 and L3) than all others. See section 8.5 for more details. A tool to measure the cache's energy consumption is RAPL [127], unfortunately at the time of writing, this is not available to our servers under test.

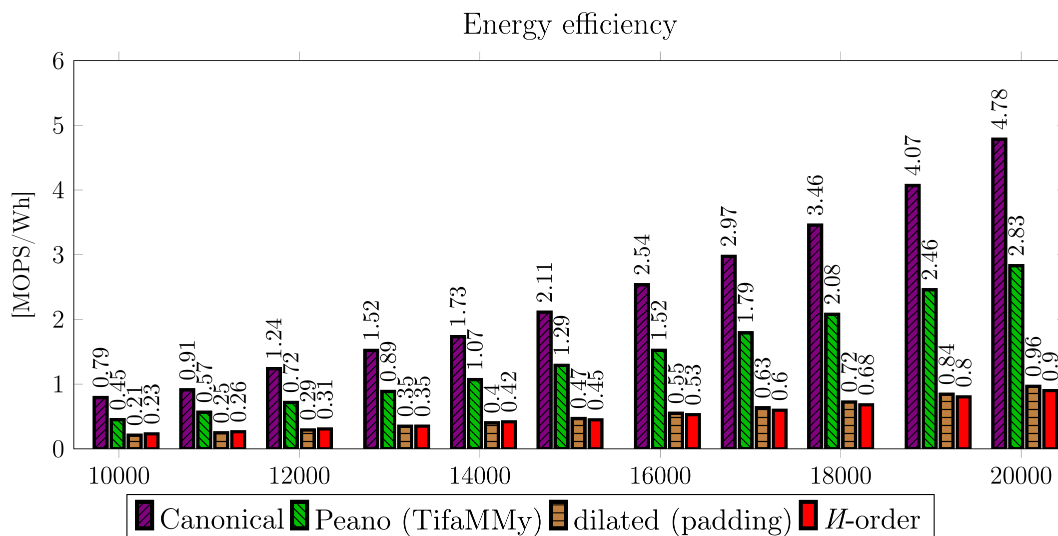


Figure 9.7: Energy efficiency of the LU decomposition on Xeon-Phi.

9.5 Conclusion

The measurement with a wattmeter is an accurate approach if we want to measure the total energy consumption of the system. However, to determine the power consumption of each subsystem (e.g., cache, stalling of CPU cycles, variation in clock frequencies) is quite challenging. An indicator that the Morton order traversal leads to energy-friendly cache access is delivered by the cache-access pattern measured by Intel© Vtune, but to ensure that the energy has been saved in a particular subsystem, we would need to test the subsystem independently from all other parts.

Chapter 10

Summary and Conclusion

In this thesis, we revisited algorithms for shared-memory environments, especially for multi- and many-core processors. We have identified and discussed various approaches in the context of MIMD and SIMD parallelism and linked these together with an efficient technique to traverse today's memory hierarchy, including registers, cache, and main memory. Our methods improve data locality, reduce cache-misses, and enhance runtime behavior.

Starting with our cache-conscious approach in chapter 3, where we tackled the K-means problem on multi-core environments, we could already recognize that there is no gold standard for all different microarchitectures. We have extended the standard K-means algorithm to an efficient algorithm designed for multi-core environments by clever usage of our cluster encoding strategy and horizontal additions and permutations of vectors in AVX2. However, the horizontal add function does not exist anymore for AVX-512, and cache-sizes and latencies keep changing for every evolving processor generation. The family of cache-oblivious algorithms addresses these drawbacks, and we extended this family in the remaining chapters of this thesis with our cache-oblivious loops. These loops replace two-nested loops in any host algorithm to traverse the memory hierarchy in an order defined by a space-filling curve to improve the data locality.

We introduced novel space-filling curves, which are based on well known space-filling curves from mathematical analysis, such as the Hilbert curve and the Morton-order curve. Our space-filling curves, such as FUR-Hilbert from chapter 5 have the advantage of filling arbitrary $n \times m$ rectangles while making only single-step moves in a recursively bisected space. The original curves are restricted to squares ($n = m$), where the side length n

must be a power of two.

In chapter 6 we applied the Hilbert curve to algorithms, such as the matrix multiplication, the K-means clustering algorithm, the algorithm by Floyd and Warshall, and the Cholesky decomposition and evaluated its performance through experimental evaluation.

The *FGF-Hilbert Join* introduced in chapter 7 is a similarity join, which optimally exploits the complete cache hierarchy by processing potential join partners in the order defined by the FGF-Hilbert curve. This curve inherits from the classical Hilbert curve its optimal locality but allows traversing regions of arbitrary size and shape. *FGF-Hilbert Join* integrates this idea into a filter-refinement algorithm. After the filter step, potential candidates are efficiently refined in FGF-Hilbert order by multiple threads in parallel using highly efficient MIMD and SIMD parallelism for the distance calculations.

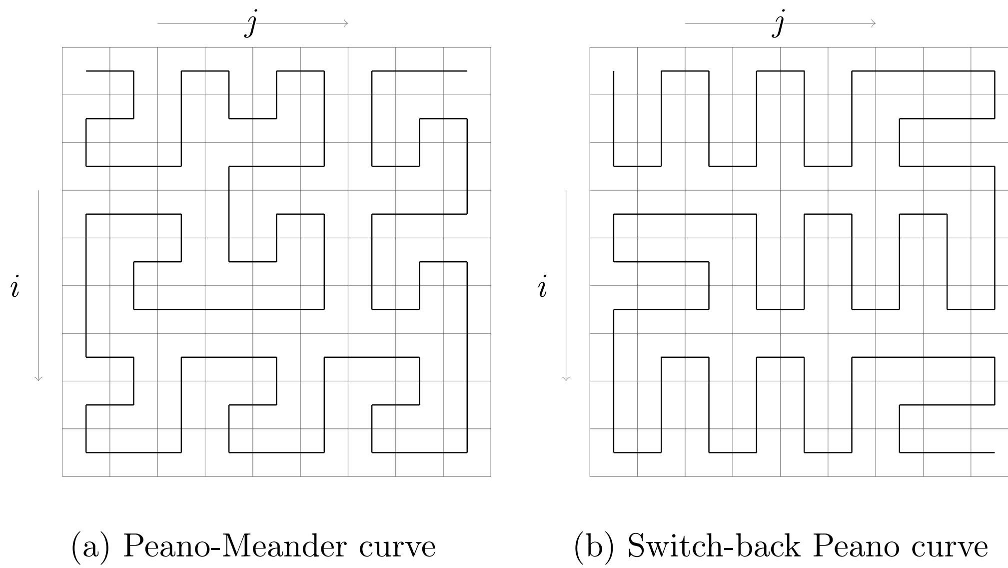
Our contribution to the Morton-order curve in chapter 8 completes our work on cache-oblivious space-filling curves. Our two different variants, the Z-order curve, and the *И*-order showed superior performance over FUR-Hilbert and the Peano curve and are not restricted to data dependencies shown for the LU decomposition.

10.1 Future work

Processors have different SIMD lengths, clock frequencies, cache-sizes, and cache latencies. Therefore, it is impossible to address all concerns and questions raised in shared memory environments. Nevertheless, exciting aspects in the context of space-filling curves that have not been covered throughout this thesis are an exploration of the memory hierarchy if other applications already occupy the cache. We believe that space-filling curves can make significant contributions in these situations.

This thesis is a proof of concept for other space-filling curves. There are still many other space-filling curves that have not been applied in the context of high-performance computing. The Peano-Meander or the Switch-back Peano curve [132] looks like a mixture of a Hilbert and Peano curve. In Figure 10.1, we visualized the third iteration of both curves.

However, in terms of data dependencies, it would be interesting to generate a new space-filling curve. One has to build a new “Leitmotiv” based on the patterns $\boxed{\sqcap}$, $\boxed{\sqcup}$, $\boxed{Z}$ or $\boxed{И}$. These patterns do not violate data dependencies observed in the most common



(a) Peano-Meander curve (b) Switch-back Peano curve

Figure 10.1: Interesting Space-Filling curves.

algorithms and serve as a good start.

Until now, very little work has been done to apply space-filling curves on the GPU. Today's Nvidia GPUs consist of memory hierarchies of up to 40 GB for the current Ampere microarchitecture. It would be interesting to test if the same effects of our work for the CPU are applicable for the GPU and its immense memory hierarchy.

Bibliography

[1] In: *Internet and Distributed Computing Systems*. Vol. 9258. Lecture Notes in Computer Science. 2015. ISBN: 978-3-319-23236-2.

[2] S. J. Ackerman, Ahmed Emam, G. Jack Lipovski, and Stanley Y. W. Su. 'Implementation of a Context-Addressed Pipeline^3 SIMD Architecture - Abstract'. In: *SIGIR Forum* 10.4 (1976), pp. 27–28.

[3] Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. 'StreamKM++: A clustering algorithm for data streams'. In: *ACM Journal of Experimental Algorithmics* 17.1 (2012).

[4] Maha Alabduljalil, Xun Tang, and Tao Yang. 'Cache-conscious performance optimization for similarity search'. In: *The 36th International ACM SIGIR conference on research and development in Information Retrieval, SIGIR '13, Dublin, Ireland - July 28 - August 01, 2013*. Ed. by Gareth J. F. Jones, Paraic Sheridan, Diane Kelly, Maarten de Rijke, and Tetsuya Sakai. ACM, 2013, pp. 713–722.

[5] Fengwei An and Hans Jürgen Mattausch. 'K-means clustering algorithm for multimedia applications with flexible HW/SW co-design'. In: *Journal of Systems Architecture - Embedded Systems Design* 59.3 (2013), pp. 155–164.

[6] E. Anderson et al. *LAPACK Users' Guide (Third Ed.)* Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8.

[7] Alexandr Andoni and Piotr Indyk. 'Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions'. In: *Commun. ACM* 51.1 (2008), pp. 117–122.

[8] Nikolaus Augsten and Michael H. Böhlen. *Similarity Joins in Relational Database Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[9] Michael Bader. 'Exploiting the Locality Properties of Peano Curves for Parallel Matrix Multiplication'. In: *Euro-Par Conference*. 2008, pp. 801–810.

[10] Michael Bader. *Space-Filling Curves - An Introduction with Applications in Scientific Computing*. Vol. 9. Texts in Computational Science and Engineering. Springer, 2013. ISBN: 978-3-642-31045-4.

[11] Michael Bader, Robert Franz, Stephan Günther, and Alexander Heinecke. 'Hardware-Oriented Implementation of Cache Oblivious Matrix Operations Based on Space-Filling Curves'. In: *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 9-12, 2007, Revised Selected Papers*. 2007, pp. 628–638.

[12] Michael Bader and Christian E. Mayer. 'Cache Oblivious Matrix Operations Using Peano Curves'. In: *PARA Workshop*. 2006, pp. 521–530.

[13] Jeroen Baert, Ares Lagae, and Philip Dutré. 'Out-of-Core Construction of Sparse Voxel Octrees'. In: *Comput. Graph. Forum* 33.6 (2014), pp. 220–227.

[14] P. Baldi, P. Sadowski, and D. Whiteson. 'Searching for exotic particles in high-energy physics with deep learning'. In: *Nature Communications* 5 (2014). Article, 4308 EP –.

[15] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 'Accelerating K-Means clustering with parallel implementations and GPU computing'. In: *HPEC Conference*. 2015, pp. 1–6.

[16] Theodore Bially. 'Space-filling curves: Their generation and their application to bandwidth reduction'. In: *IEEE Trans. Information Theory* 15 (1969), pp. 658–664.

[17] Markus Bläser. 'Fast Matrix Multiplication'. In: *Theory of Computing, Graduate Surveys* 5 (2013), pp. 1–60.

[18] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. 'High Performance Clustering Based on the Similarity Join'. In: *CIKM*. 2000, pp. 298–305.

[19] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. 'Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data'. In: *SIGMOD Conf. 2001*. 2001, pp. 379–388.

[20] Christian Böhm and Florian Krebs. 'The k-Nearest Neighbour Join: Turbo Charging the KDD Process'. In: *Knowl. Inf. Syst.* 6.6 (2004), pp. 728–749.

[21] Christian Böhm and Hans-Peter Kriegel. 'A Cost Model and Index Architecture for the Similarity Join'. In: *ICDE*. 2001, pp. 411–420.

[22] Christian Böhm, Robert Noll, Claudia Plant, and Andrew Zherdin. 'Indexsupported Similarity Join on Graphics Processors'. In: *Datenbanksysteme in Business, Technologie und Web BTW 2009*. 2009, pp. 57–66.

[23] Christian Böhm, Martin Perdacher, and Claudia Plant. 'A Novel Hilbert Curve for Cache-locality Preserving Loops'. In: *IEEE Transactions on Big Data* (2018), pp. 1–14. ISSN: 2332-7790.

[24] Christian Böhm, Martin Perdacher, and Claudia Plant. 'Cache-oblivious loops based on a novel space-filling curve'. In: *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*. IEEE Computer Society, 2016, pp. 17–26.

[25] Christian Böhm, Martin Perdacher, and Claudia Plant. 'Multi-core K-means'. In: *Proceedings of the 2017 SIAM International Conference on Data Mining, Houston, Texas, USA, April 27-29, 2017*. Ed. by Nitesh V. Chawla and Wei Wang. SIAM, 2017, pp. 273–281.

[26] Douglas C. Bossen, Joel M. Tendler, and Kevin Reick. 'Power4 System Design for High Reliability'. In: *IEEE Micro* 22.2 (2002), pp. 16–24.

[27] Greg Breinholt and Christoph Schierz. 'Algorithm 781: Generating Hilbert's Space-filling Curve by Recursion'. In: *ACM Trans. Math. Softw.* 24.2 (June 1998), pp. 184–189. ISSN: 0098-3500.

[28] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 'Efficient Processing of Spatial Joins Using R-Trees'. In: *SIGMOD Conf. 1993*. 1993, pp. 237–246.

[29] Brent Bryan, Frederick Eberhardt, and Christos Faloutsos. 'Compact Similarity Joins'. In: *ICDE*. 2008, pp. 346–355.

[30] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 'A class of parallel tiled linear algebra algorithms for multicore architectures'. In: *Parallel Computing* 35.1 (2009), pp. 38–53.

[31] Y. Dora Cai, Rabindra Robby Ratan, Cuihua Shen, and Jay Alameda. 'Grouping game players using parallelized k-means on supercomputers'. In: *XSEDE Conference*. 2015, 10:1–10:7.

[32] Georg Cantor. 'Ueber unendliche, lineare Punktmannichfaltigkeiten'. In: *Mathematische Annalen* 17.3 (1880), pp. 355–358.

[33] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. 'Recursive Array Layouts and Fast Matrix Multiplication'. In: *IEEE Trans. Parallel Distrib. Syst.* 13.11 (2002), pp. 1105–1123.

[34] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, and Mithuna Thottethodi. 'Recursive Array Layouts and Fast Parallel Matrix Multiplication'. In: *Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '99, Saint-Malo, France, June 27-30, 1999.* Ed. by Gary L. Miller and Vijaya Ramachandran. ACM, 1999, pp. 222–231.

[35] Lu Chen, Yunjun Gao, Xinhan Li, Christian S. Jensen, and Gang Chen. 'Efficient Metric Indexing for Similarity Search and Similarity Joins'. In: *IEEE Trans. Knowl. Data Eng.* 29.3 (2017), pp. 556–571.

[36] Ningtao Chen, Nengchao Wang, and Baochang Shi. 'A new algorithm for encoding and decoding the Hilbert order'. In: *Softw., Pract. Exper.* 37.8 (2007), pp. 897–908.

[37] Tse-Wei Chen, Chih-Hao Sun, Hsiao-Hang Su, Shao-Yi Chien, D. Deguchi, I. Ide, and H. Murase. 'Power-Efficient Hardware Architecture of K-Means Clustering With Bayesian-Information-Criterion Processor for Multimedia Processing Applications'. In: *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on* 1.3 (2011), pp. 357–368. ISSN: 2156-3357.

[38] Jaeyoung Choi, James Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. 'ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance'. In: *Applied Parallel Computing, Computations in Physics, Chemistry and Engineering Science, Second International Workshop, PARA '95, Lyngby, Denmark, August 21-24, 1995, Proceedings.* 1995, pp. 95–106.

[39] Noam Chomsky. 'Three models for the description of language'. In: *IRE Trans. Inf. Theory* 2.3 (1956), pp. 113–124.

[40] Kuo-Liang Chung, Yi-Luen Huang, and Yau-Wen Liu. 'Efficient algorithms for coding Hilbert curve of arbitrary-sized image and application to window query'. In: *Inf. Sci.* 177.10 (2007), pp. 2130–2151.

[41] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 'M-tree: An Efficient Access Method for Similarity Search in Metric Spaces'. In: *VLDB'97.* 1997, pp. 426–435.

[42] William J. Dally, James D. Balfour, David Black-Schaffer, James Chen, R. Curtis Harting, Vishal Parikh, JongSoo Park, and David Sheffield. 'Efficient Embedded Computing'. In: *Computer* 41.7 (2008), pp. 27–32.

[43] Dong Deng, Yufei Tao, and Guoliang Li. 'Overlap Set Similarity Joins with Theoretical Guarantees'. In: *SIGMOD Conf. 2018*. 2018, pp. 905–920.

[44] Jens-Peter Dittrich and Bernhard Seeger. 'GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces'. In: *SIGKDD*. 2001, pp. 47–56.

[45] Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. 'D-Index: Distance Searching Index for Metric Data Sets'. In: *Multimedia Tools Appl.* 21.1 (2003), pp. 9–33.

[46] Vlastislav Dohnal, Claudio Gennaro, and Pavel Zezula. 'Similarity Join in Metric Spaces Using eD-Index'. In: *DEXA 2003*. 2003, pp. 484–493.

[47] Jack Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. 'A set of level 3 basic linear algebra subprograms'. In: *ACM Trans. Math. Softw.* 16.1 (1990), pp. 1–17.

[48] Ralph Duncan. 'A Survey of Parallel Computer Architectures'. In: *Computer* 23.2 (1990), pp. 5–16.

[49] Charles Elkan. 'Using the Triangle Inequality to Accelerate k-Means'. In: *ICML Conference*. 2003, pp. 147–153.

[50] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 5th edition, 2006.

[51] Miguel Ferreira, Nuno Roma, and Luís M. S. Russo. 'Cache-Oblivious parallel SIMD Viterbi decoding for sequence search in HMMER'. In: *BMC Bioinformatics* 15 (2014), p. 165.

[52] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. 'Set Similarity Joins on MapReduce: An Experimental Survey'. In: *PVLDB* 11.10 (2018), pp. 1110–1122.

[53] Michael J Flynn. 'Very high-speed computing systems'. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[54] F.D. Fracchia, P. Prusinkiewicz, and A. Lindenmayer. *Synthesis of Space-filling Curves on the Square Grid*. Amsterdam, The Netherlands: Elsevier Sci. Pub. B. V., 1991, pp. 341–366. ISBN: 9780773101715.

[55] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. 'Cache-Oblivious Algorithms'. In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. 1999, pp. 285–298.

[56] Irene Gargantini. 'An Effective Way to Represent Quadtrees'. In: *Commun. ACM* 25.12 (1982), pp. 905–910.

[57] David K. Gifford and Alfred Z. Spector. 'The Space Shuttle Primary Computer System'. In: *Commun. ACM* 27.9 (1984), pp. 872–900.

[58] Ali Hadian and Saeed Shahrivari. 'High performance parallel k-means clustering for disk-resident datasets on multi-core CPUs'. In: *The Journal of Supercomputing* 69.2 (2014), pp. 845–863.

[59] Md Rakib Hasan and R. Clint Whaley. 'Effectively Exploiting Parallel Scale for All Problem Sizes in LU Factorization'. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. 2014, pp. 1039–1048.

[60] Bingsheng He, Yinan Li, Qiong Luo, and Dongqing Yang. 'EaseDB: a cache-oblivious in-memory query processor'. In: *SIGMOD Conf. 2007*. 2007, pp. 1064–1066.

[61] Alexander Heinecke and Carsten Trinitis. 'Cache-oblivious matrix algorithms in the age of multicores and many cores'. In: *Concurrency and Computation: Practice and Experience* 27.9 (2010), pp. 2215–2234.

[62] Alexander Heinecke and Carsten Trinitis. 'Making TifaMMy fit for tomorrow: Towards future shared memory systems and beyond'. In: *2011 International Conference on High Performance Computing & Simulation, HPCS 2012, Istanbul, Turkey, July 4-8, 2011*. 2011, pp. 517–524.

[63] Duong Van Hieu and Phayung Meesad. 'Fast K-Means Clustering for Very Large Datasets Based on MapReduce Combined with a New Cutting Method'. In: *KSE conference*. 2014, pp. 287–298.

[64] David Hilbert. 'Über die stetige Abbildung einer Linie auf ein Flächenstück'. In: *Mathematische Annalen* 38 (1891).

[65] ThienLuan Ho, Seungrohk Oh, and Hyunjin Kim. 'New algorithms for fixed-length approximate string matching and approximate circular string matching under the Hamming distance'. In: *The Journal of Supercomputing* 74.5 (2018), pp. 1815–1834.

[66] *Intel 64 and IA-32 Architectures Optimization Reference Manual.*

[67] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2Nd Edition.* 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN: 0128091940, 9780128091944.

[68] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. 'Product Quantization for Nearest Neighbor Search'. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 33.1 (2011), pp. 117–128.

[69] Dmitri V. Kalashnikov. 'Super-EGO: fast multi-dimensional similarity join'. In: *VLDB J.* 22.4 (2013), pp. 561–585.

[70] Dmitri V. Kalashnikov and Sunil Prabhakar. 'Fast similarity join for multi-dimensional data'. In: *Inf. Syst.* 32.1 (2007), pp. 160–177.

[71] Dmitri V. Kalashnikov and Sunil Prabhakar. 'Similarity Join for Low-and High-Dimensional Data'. In: *(DASFAA '03).* 2003, pp. 7–16.

[72] Helge van Koch. 'Sur une courbe continue sans tangente, obtenue par une construction geometrique elementaire/Van Koch Helge'. In: *Arkiv for Matematik* 1 (1904), pp. 681–704.

[73] Nick Koudas and Kenneth C. Sevcik. 'High Dimensional Similarity Joins: Algorithms and Performance Evaluation'. In: *IEEE Trans. Knowl. Data Eng.* 12.1 (2000), pp. 3–18.

[74] Susana Ladra, Oscar Pedreira, José Duato, and Nieves R. Brisaboa. 'Exploiting SIMD Instructions in Current Processors to Improve Classical String Algorithms'. In: *ADBIS conference.* 2012, pp. 254–267.

[75] Shigang Li, Yunquan Zhang, and Torsten Hoefler. 'Cache-Oblivious MPI All-to-All Communications Based on Morton Order'. In: *IEEE Trans. Parallel Distrib. Syst.* 29.3 (2018), pp. 542–555.

[76] Ye Li, Jian Wang, and Leong Hou U. 'Multidimensional Similarity Join Using MapReduce'. In: *Web-Age Information Management.* 2016, pp. 457–468.

[77] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. 'Speeding up k-Means algorithm by GPUs'. In: *J. Comput. Syst. Sci.* 79.2 (2013), pp. 216–229.

[78] M. Lichman. *UCI Machine Learning Repository.* 2013.

[79] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 'A Fast Similarity Join Algorithm Using Graphics Processing Units'. In: *ICDE*. 2008, pp. 1111–1120.

[80] Jonathan Lifflander, Phil Miller, Ramprasad Venkataraman, Anshu Arya, Laxmikant V. Kalé, and Terry Jones. 'Mapping Dense LU Factorization on Multicore Supercomputer Nodes'. In: *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012*. 2012, pp. 596–606.

[81] Aristid Lindenmayer. 'Mathematical models for cellular interaction in development: Parts I and II.' In: *Journal of Theoretical Biology* 18 (1968).

[82] K. Patrick Lorton and David S. Wise. 'Analyzing block locality in Morton-order and Morton-hybrid matrices'. In: *SIGARCH Computer Architecture News* 35.4 (2007), pp. 6–12.

[83] Youzhong Ma, Shijie Jia, and Yongxin Zhang. 'A novel approach for high-dimensional vector similarity join query'. In: *Concurrency and Computation: Practice and Experience* 29.5 (2017).

[84] Tobias Maier, Peter Sanders, and Jochen Speck. 'Locality Aware DAG-Scheduling for LU-Decomposition'. In: *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*. 2015, pp. 82–92.

[85] Samuel McCauley and Francesco Silvestri. 'Adaptive MapReduce Similarity Joins'. In: *SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. 2018, 4:1–4:4.

[86] Panagiotis D. Michailidis and Konstantinos G. Margaritis. 'Implementing Parallel LU Factorization with Pipelining on a MultiCore Using OpenMP'. In: *13th IEEE International Conference on Computational Science and Engineering, CSE 2010, Hong Kong, China, December 11-13, 2010*. 2010, pp. 253–260.

[87] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. 'Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection'. In: *Network and Distributed System Security Symposium, NDSS*. 2018.

[88] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. 'Cache-oblivious ray reordering'. In: *ACM Trans. Graph.* 29.3 (2010).

[89] G. H. Morton. 'A computer oriented geodetic data base and a new technique in file sequencing'. In: 1966.

[90] Ernst Naschenweng. 'A Cache Optimized Implementation of the Floyd-Warshall Algorithm Using Hardware Acceleration Through Intel Intrinsics'. Bachelor's Thesis. Währinger Straße 29, 1090 Vienna: University of Vienna, Research Group of Data Mining, 2018.

[91] Saul B. Needleman and Christian D. Wunsch. 'A general method applicable to the search for similarities in the amino acid sequence of two proteins'. In: *Journal of Molecular Biology* 48.3 (1970), pp. 443 –453. ISSN: 0022-2836.

[92] Giap Nguyen, Patrick Franco, and Jean-Marc Ogier. 'Space-Filling Curve for Image Dynamical Indexing'. In: *Computer and Information Sciences III - 27th International Symposium on Computer and Information Sciences, Paris, France, October 3-4, 2012*. Ed. by Erol Gelenbe and Ricardo Lent. Springer, 2012, pp. 311–319.

[93] Anthony E. Nocentino and Philip J. Rhodes. 'Optimizing memory access on GPUs using morton order indexing'. In: *Proceedings of the 48th Annual Southeast Regional Conference, 2010, Oxford, MS, USA, April 15-17, 2010*. Ed. by H. Conrad Cunningham, Paul Ruth, and Nicholas A. Kraft. ACM, 2010, p. 18.

[94] Jack A. Orenstein. 'Spatial Query Processing in an Object-oriented Database System'. In: *SIGMOD Conf. 1986*. 1986, pp. 326–336.

[95] Rasmus Pagh, Ninh Pham, Francesco Silvestri, and Morten Stöckel. 'I/O-Efficient Similarity Join'. In: *Algorithmica* 78.4 (2017), pp. 1263–1283.

[96] Szilárd Páll and Berk Hess. 'A flexible algorithm for calculating pair interactions on SIMD architectures'. In: *Computer Physics Communications* 184.12 (2013), pp. 2641–2650.

[97] Nicolas Papernot and Patrick D. McDaniel. 'Deep k-Nearest Neighbors: Towards Confident, Interpretable and Robust Deep Learning'. In: *CoRR* abs/1803.04765 (2018). arXiv: 1803.04765.

[98] Rodrigo Paredes and Nora Reyes. 'Solving similarity joins and range queries in metric spaces with the list of twin clusters'. In: *J. Discrete Algorithms* 7.1 (2009), pp. 18–35.

[99] Jeff Parkhurst, John A. Darringer, and Bill Grundmann. 'From single core to multi-core: preparing for a new exponential'. In: *2006 International Conference on Computer-Aided Design, ICCAD 2006, San Jose, CA, USA, November 5-9, 2006*. Ed. by Soha Hassoun. ACM, 2006, pp. 67–72.

[100] Filip Pawlowski, Bora Uçar, and Albert-Jan Yzelman. 'A multi-dimensional Morton-ordered block storage for mode-oblivious tensor computations'. In: *J. Comput. Science* 33 (2019), pp. 34–44.

[101] Peano. 'Sur une courbe, qui remplit toute une aire plane'. In: *Mathematische Annalen* 36 (1890), pp. 157–160.

[102] Spencer S. Pearson and Yasin N. Silva. 'Index-Based R-S Similarity Joins'. In: *SISAP*. 2014, pp. 106–112.

[103] Dan Pelleg and Andrew Moore. 'X-means: Extending K-means with Efficient Estimation of the Number of Clusters'. In: *ICML conference*. 2000, pp. 727–734.

[104] Martin Perdacher, Claudia Plant, and Christian Böhm. 'Cache-oblivious High-performance Similarity Join'. In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 2019, pp. 87–104.

[105] Martin Perdacher, Claudia Plant, and Christian Böhm. 'Improved Data Locality Using Morton-order Curve on the Example of LU Decomposition'. In: *2020 IEEE International Conference on Big Data, BigData 2020, Atlanta, GA, USA, December 10-13, 2020*. Accepted for publication. Dec. 2020.

[106] Loren K. Platzman and John J. Bartholdi III. 'Spacefilling curves and the planar travelling salesman problem'. In: *J. ACM* 36.4 (1989), pp. 719–737.

[107] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes: the art of scientific computing, 3rd Edition*. Cambridge University Press, 2007.

[108] Hans Sagan. *Space-filling curves*. Universitext Series. Springer-Verlag, 1994. ISBN: 9780387942650.

[109] Hanan Samet. 'The Quadtree and Related Hierarchical Data Structures'. In: *ACM Comput. Surv.* 16.2 (1984), pp. 187–260.

[110] Donovan A. Schneider and David J. DeWitt. 'A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment'. In: *SIGMOD Conf. 1989*. 1989, pp. 110–121.

[111] Ayon Sen, A. S. M. Sohidull Islam, and Md. Yusuf Sarwar Uddin. 'MARQUES: Distributed multi-attribute range query solution using space filling curve on DTHs'. In: *International Conference on Networking Systems and Security, NSysS 2015, Dhaka, Bangladesh, January 5-7, 2015*. IEEE, 2015, pp. 1–9.

[112] Binanda Sengupta and Abhijit Das. *Use of SIMD-Based Data Parallelism to Speed up Sieving in Integer-Factoring Algorithms*. Cryptology ePrint Archive, Report 2015/044. 2015.

[113] Saeed Shahrivari and Saeed Jalili. 'Single-pass and linear-time k-means clustering based on MapReduce'. In: *Inf. Syst.* 60 (2016), pp. 1–12.

[114] Zeyuan Shang, Yaxiao Liu, Guoliang Li, and Jianhua Feng. 'K-Join: Knowledge-Aware Similarity Join'. In: *ICDE*. 2017, pp. 23–24.

[115] Shubham Sharma, K. V. S. Hari, and Geert Leus. 'Space Filling Curves for MRI Sampling'. In: *2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020*. IEEE, 2020, pp. 1115–1119.

[116] Michael Shindler, Alex Wong, and Adam Meyerson. 'Fast and Accurate k-means For Large Datasets'. In: *NIPS conference*. 2011, pp. 2375–2383.

[117] H. J. S. Smith. *On the Integration of Discontinuous Functions*. Nov. 1874.

[118] Allan Stisen, Henrik Blunck, Sourav Bhattacharya, Thor Siiger Prentow, Mikkel Baun Kjærgaard, Anind K. Dey, Tobias Sonne, and Mads Møller Jensen. 'Smart Devices are Different: Assessing and MitigatingMobile Sensing Heterogeneities for Activity Recognition'. In: *Embedded Networked Sensor Systems, SenSys*. 2015, pp. 127–140.

[119] Mehul Tikekar, Chao-Tsung Huang, Chiraag Juvekar, Vivienne Sze, and Anantha P. Chandrakasan. 'A 249-Mpixel/s HEVC Video-Decoder Chip for 4K Ultra-HD Applications'. In: *IEEE J. Solid State Circuits* 49.1 (2014), pp. 61–72.

[120] Po-An Tsai, Nathan Beckmann, and Daniel Sánchez. 'Jenga: Software-Defined Cache Hierarchies'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 2017, pp. 652–665.

[121] Satori Tsuzuki and Takayuki Aoki. 'Effective Dynamic Load Balance using Space-Filling Curves for Large-Scale SPH Simulations on GPU-rich Supercomputers'. In: *7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA@SC 2016, Salt Lake, UT, USA, November 14, 2016*. IEEE Computer Society, 2016, pp. 1–8.

[122] Vinod Valsalam and Anthony Skjellum. 'A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels'. In: *Concurr. Comput. Pract. Exp.* 14.10 (2002), pp. 805–839.

[123] David W. Walker. 'Morton ordering of 2D arrays for efficient access to hierarchical memory'. In: *Int. J. High Perform. Comput. Appl.* 32.1 (2018), pp. 189–203.

[124] Jiajun Wang, Prakash Ramrakhyani, Wendy Elsasser, and Lizy Kurian John. 'Reducing Data Movement and Energy in Multilevel Cache Hierarchies without Losing Performance: Can you have it all?' In: *28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019*. IEEE, 2019, pp. 383–394.

[125] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. 'Scalable all-pairs similarity search in metric spaces'. In: *SIGKDD*. 2013, pp. 829–837.

[126] Stephen Warshall. 'A Theorem on Boolean Matrices'. In: *J. ACM* 9 (1962), pp. 11–12.

[127] Vincent M. Weaver, Matt Johnson, Kiran Kasichayanula, James Ralph, Piotr Luszczek, Daniel Terpstra, and Shirley Moore. 'Measuring Energy and Power with PAPI'. In: *41st International Conference on Parallel Processing Workshops, ICPPW 2012, Pittsburgh, PA, USA, September 10-13, 2012*. IEEE Computer Society, 2012, pp. 262–268.

[128] R. Clinton Whaley, Antoine Petitet, and Jack J. Dongarra. 'Automated empirical optimizations of software and the ATLAS project'. In: *Parallel Computing* 27.1-2 (2001), pp. 3–35.

[129] Thomas Wirtz, Rong Ge, Ziliang Zong, and Zizhong Chen. 'Power and energy characteristics of MapReduce data movements'. In: *International Green Computing Conference, IGCC 2013, Arlington, VA, USA, June 27-29, 2013, Proceedings*. 2013, pp. 1–7.

[130] David S. Wise. 'Ahnentafel Indexing into Morton-Ordered Arrays, or Matrix Locality for Free'. In: *Euro-Par 2000, Parallel Processing, 6th International Euro-Par Conference, Munich, Germany, August 29 - September 1, 2000, Proceedings*. 2000, pp. 774–783.

[131] Rongteng Wu and Xiaohong Xie. 'Two-Stage Column Block Parallel LU Factorization Algorithm'. In: *IEEE Access* 8 (2020), pp. 2645–2655.

[132] W Wunderlich. 'Über Peano-Kurven.' In: *Elemente der Mathematik* 28 (1973), pp. 1–10.

[133] Chuan Xiao, Wei Wang, and Xuemin Lin. 'Ed-Join: an efficient algorithm for similarity joins with edit distance constraints'. In: *PVLDB* 1.1 (2008), pp. 933–944.

[134] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 'Efficient similarity joins for near-duplicate detection'. In: *ACM Trans. Database Syst.* 36.3 (2011), 15:1–15:41.

[135] Yujie Xu, Wenyu Qu, Zhiyang Li, Changqing Ji, Yuanyuan Li, and Yinan Wu. 'Fast Scalable k-means++ Algorithm with MapReduce'. In: *ICA3PP conference.* 2014, pp. 15–28.

[136] Ichitaro Yamazaki and Xiaoye S. Li. 'New Scheduling Strategies and Hybrid Programming for a Parallel Right-looking Sparse LU Factorization Algorithm on Multicore Cluster Systems'. In: *26th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2012, Shanghai, China, May 21-25, 2012.* 2012, pp. 619–630.

[137] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. 'A Generic Method for Accelerating LSH-Based Similarity Join Processing'. In: *IEEE Trans. Knowl. Data Eng.* 29.4 (2017), pp. 712–726.

[138] Xianyi Zhang, Qian Wang, and Yunquan Zhang. 'Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor'. In: *18th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2012, Singapore, December 17-19, 2012.* 2012, pp. 684–691.

[139] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. 'A General SIMD-Based Approach to Accelerating Compression Algorithms'. In: *ACM Trans. Inf. Syst.* 33.3 (2015), 15:1–15:28.

[140] Weijie Zhao, Florin Rusu, Bin Dong, and Kesheng Wu. 'Similarity Join over Array Data'. In: *SIGMOD Conf. 2016.* 2016, pp. 2007–2022.

Appendix A

Further Experiments

A.1 Runtime Performance on Morton-order Curves

N	*Runtime (pext)*	*GFLOPS (pext)*	*Runtime (tzcnt)*	*GFLOPS (tzcnt)*
1080	0.02528560	99.80929	0.02567013	98.21039
2040	0.05674588	299.32351	0.05767773	294.52095
3000	0.10799772	500.18867	0.11159680	483.99828
4080	0.21694396	626.17247	0.22232593	611.21640
5040	0.33183284	771.90906	0.35442280	723.40718
6000	0.52795712	818.31903	0.55517820	778.81669
7080	0.83910536	847.86541	0.85860920	827.11213
8040	1.22086588	851.42188	1.22177987	850.80200
9000	1.75569396	830.45738	1.80084493	809.93762
10080	2.41217496	849.20357	2.50227400	819.12002
11040	3.14761644	854.99675	3.25397660	827.55675
12000	4.00843500	862.19411	4.15032567	833.21511
13080	5.13225008	872.06854	5.31713333	842.18406
14040	6.34974896	871.73409	6.58571127	840.84710
15000	7.70795608	875.73250	7.97758180	846.55658
16080	9.49125264	876.13560	9.84341187	844.82968
17040	11.71310596	844.84022	12.53866927	789.42072
18000	13.80301960	845.05089	15.05740027	775.90689
19080	16.30293752	852.12832	18.05350733	772.58042
20040	18.78117756	857.04377	20.96695720	771.67350

Table A.1: Runtime on two different Morton order implementations tested on matrix multiplication.

N	*Runtime (pext)*	*GFLOPS (pext)*	*Runtime (tzcnt)*	*GFLOPS (tzcnt)*
1080	0.0565760	15.05235	0.0597580	14.09585
2040	0.1470210	38.55971	0.1443688	39.28681
3000	0.2188178	82.38057	0.2225354	80.97075
4080	0.3319146	136.52417	0.3384730	133.88474
5040	0.4607914	185.35968	0.4660542	183.26344
6000	0.5827974	247.21604	0.5889154	244.64207
7080	0.7519278	314.81322	0.7566930	312.83242
8040	0.9773798	354.63445	0.9869642	351.19145
9000	1.2025738	404.27311	1.1984746	405.67026
10080	1.5352118	444.90485	1.5366988	444.47598
11040	1.8739252	478.83964	1.8717652	479.38502
12000	2.2783454	505.75729	2.2795128	505.50180
13080	2.7932188	534.23548	2.8086294	531.31211
14040	3.3228154	555.39365	3.3426794	552.09495
15000	3.8951066	577.76800	3.8972954	577.44179
16080	4.6595276	594.98591	4.6557162	595.47314
17040	5.4327788	607.26237	5.4261852	607.99708
18000	6.2750236	619.71103	6.2708224	620.12559
19080	7.4001814	625.85387	7.4479432	621.86005
20040	8.9609248	599.65432	11.4387290	470.48430

Table A.2: Runtime on two different Morton order implementations tested on LU decomposition.