Cache-Oblivious Nested Loops Based on Hilbert Curves

João Nuno Estevão Fidalgo Ferreira Alves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Alexandre Paulo Lourenço Francisco
Prof. Luís Manuel Silveira Russo

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Alexandre Paulo Lourenço Francisco
Member of the Committee: Prof. Luís Jorge Brás Monteiro Guerra e Silva

October 2019

Acknowledgments

I would like to express my gratitude to Prof. Alexandre Francisco and Prof. Luís Russo for their insight, patience and mentorship. Without their knowledge it would be impossible to realize this thesis. I would also like to thank my parents for all the support and for making me want to be the best version of myself. Last but not the least, I would like to thank to my girlfriend for being supportive and caring during one of the most stressful moments of my life.

Abstract

Many fields of computer science, especially data science and artificial intelligence, are becoming challenged with an immeasurable amount of data to process. The recent work developed by *Böhm et al.* [1] presents us a novelty in the field of the algorithms, a Cache-Oblivious Nested For-Loop. This algorithm allows programmers to optimize the cache behaviour of nested for-loops, without any need of knowledge about the CPU cache specifications. In this thesis we will provide an efficient alternative approach to this algorithm, which we called XOR-Hilbert. Our algorithm presents a linear growth-rate of memory and time. The use of memory makes our algorithm stateful, thus allowing the re-utilization of previously computed Hilbert Space-Filling Curves. Which allows the time required to compute another iteration of this curve to be amortized.

Keywords

cache-oblivious algorithms; Hilbert curve; space-filling curves; cache-oblivious for-loop.

Resumo

Diversos campos da engenharia informática, mais especificamente ciência de dados e inteligência artificial, apresentam dificuldades em processar quantidades imesuráveis de dados. O trabalho desenvolvido por *Böhm et al.* [1] apresenta-nos uma novidade no campo dos algoritmos, um ciclo aninhado que optimiza a taxa de faltas da cache, sem qualquer conhecimento sobre as suas especificações. Nesta tese será apresentado uma abordagem alternativa e eficiente chamada XOR-Hilbert. O nosso algoritmo apresenta uma taxa de crescimento linear, tanto para o tempo de execução como para memória utilizada. O uso de memória desta abordagem possibilita a reutilização de curvas já computadas, amortizando assim o tempo de execução deste algoritmo para chamadas repetidas da mesma curva.

Palavras Chave

curva de Hilbert; algorítmos *cache-oblivious*; curvas de preenchimento de espaço; ciclos for baseado na curva de Hilbert;

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Contents

Beneath most fields of exact sciences and engineering, and in particular of computer science, such as data science, machine learning, and graph theory, one will find matrices to be one of the inevitable primitive types, whose operations allow the computation of other more complex algorithms. This lead us to the reasonable conclusion that by optimizing these primitive operations, more complex operations that make usage of these primitives will also be optimized. It is also known that the implementation of algorithms, in a physical machine, presents an overhead in comparison to its mathematical counterpart, due to hardware limitations, even if the run-time complexity is the same in both cases. In order to reduce this cost it is required from the programmer behalf to make smart usage of the computer components. This can be hard to do in most cases, since high-level programming languages tend to have semantics closer to natural language, thus hiding hardware details from the programmer. In 1996, in *Massachusetts Institute of Technology*, a new concept was conceived by Charles E. Leiserson. The notion of Cache-Oblivious algorithms. One of the most costly hardware operations is reading and writing from and to memory. To minimize this cost the computer processor contains a cache, that stores recently fetched blocks from memory, for fast access to data without having to fetch this information every time it is needed. It is also important to note that data travels from main memory to cache in blocks, i.e. if data is fetched from memory address, the data contained by adjacent memory addresses will also be loaded into cache. This allows the cache to take advantage of spatial locality of data in memory, and amortizes the latency time of reading operations from main memory. The previously referenced concept, Cache-Obliviousness, represents a set of algorithms whose cache accesses are optimized, without any knowledge of Cache characteristics, such as line size and memory hierarchy, thus making this algorithms extremely portable. In 2016 *Böhm et al.* [1] presented a novel Space-Filling Curve(SFC) able to traverse matrices with any given dimension. Upon this work it was build the first Cache-Oblivious Loop, called FUR-Hilbert. This nested-loop, based on the Hilbert Space-Filling Curve(HSFC), preserves cache data locality better than ordinary orders of traversal, such as row-major and column-major orders.

Algorithm 1.1: Row-Major Order Multiplication

```
Input: $A$ and $B^\top$
Output: $C$
begin
    $C[4,4]$;
    $i := 0$;
    $j := 0$;
    while $i < 4$ do
        while $j < 4$ do
            $C_{i,j} := \sum_{k=0}^{3}(A_{i,k} \times B^\top_{j,k})$;
            $j := j + 1$;
            $i := i + 1$;
    return $C$;
```

Algorithm 1.2: HSFC based Multiplication

```
Input: $A$ and $B^\top$
Output: $C$
begin
    $C[4,4]$;
    $i := 0$;
    $j := 0$;
    while $h < 16$ do
        $(i,j) := H^{-1}(h)$;
        $C_{i,j} := \sum_{k=0}^{3}(A_{i,k} \times B^\top_{j,k})$;
        $j := j + 1$;
        $i := i + 1$;
    return $C$;
```

1.1 Motivation

There is no better way to explain the motive that leads us to perform this study than comparing two different approaches that perform matrix multiplications. The first approach is based on a row-major order traversal, as depicted by Algorithm 1.1. While the path of traversal of the second approach is based on the second iteration of an HSFC, performed by Algorithm 1.2.

Example 1.1. *Row-Major Order V.S. HSFC Based Matrix Multiplication*
In order to provide a simplified comparison between these two approaches, the following conditions will be established:

- *Let A, B and C be 4×4 matrices. Both of the matrices are stored in memory using row-major order.*
- *Let i and j be the the row and column index, where $i, j \in \mathbb{N}_0$ and are bounded by $0 \leq i, j < 4$, $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$ represent entry (i, j) of matrix A, B, or C, respectively.*
- *Assume our cache in use has only 2 lines, these lines store the data contained by a block. Each block has enough space to contain 8 integers, meaning each line of the cache can contain 2 complete consecutive rows of A, or columns of B.*
- *Cache replacement policy is optimal. If no cache line is free, the line to be replaced with a new block is the one that will be used further in the future.*
- *The result of the product between matrix A and B will be stored in matrix C, However for simplicity sake we will disregard memory accesses to C.*

In order to compute $C_{i,j}$ row i of A and row j of $B^\top$ must be fetched. In algorithm 1.2, $H^{-1}(h)$ represents the Hilbert inverse, a function that establishes a direct mapping between the h^{th} matrix entry to be visited and its pair of coordinates (i, j). Tables 1.1 describe the cache behaviour of row-major order (left-side table) and HSFC based approach (right-side table), respectively. The rows of these tables are sorted by order of traversal.

The approach based on a row-major order traversal will result in 10 cache-misses. On the other hand the approach based on HSFC results in 5 blocks to be fetched from main memory. Typically the amount of time required to read data from main memory is around 70 ns. While loading data from cache only incurs in 2 ns, as stated in [2]. Implying Algorithm 1.1 has an overhead of $(22 \times 2) + (10 \times 70) = 744$ ns. While Algorithm 1.2, based on space-filling curves, has an overhead of $(27 \times 2) + (5 \times 70) = 404$ ns. It is important to note that this overhead is only associated with memory accesses, thus representing only a fraction of the total run-time these approaches impose.

Table 1.1: Row-major order versus Hilbert order traversal.

	Rows Needed	Rows in Cache	#Cumulative Cache-Misses		Rows Needed	Rows in Cache	#Cumulative Cache-Misses
$C_{0,0}$	$A_0, B_0^\top$	-	2	$C_{0,0}$	$A_0, B_0^\top$	-	2
$C_{0,1}$	$A_0, B_1^\top$	$A_0, A_1, B_0^\top, B_1^\top$	2	$C_{0,1}$	$A_0, B_1^\top$	$A_0, A_1, B_0^\top, B_1^\top$	2
$C_{0,2}$	$A_0, B_2^\top$	$A_0, A_1, B_0^\top, B_1^\top$	3	$C_{1,1}$	$A_1, B_1^\top$	$A_0, A_1, B_0^\top, B_1^\top$	2
$C_{0,3}$	$A_0, B_3^\top$	$A_0, A_1, B_2^\top, B_3^\top$	3	$C_{1,0}$	$A_1, B_0^\top$	$A_0, A_1, B_0^\top, B_1^\top$	2
$C_{1,0}$	$A_1, B_0^\top$	$A_0, A_1, B_2^\top, B_3^\top$	4	$C_{2,0}$	$A_2, B_0^\top$	$A_0, A_1, B_0^\top, B_1^\top$	3
$C_{1,1}$	$A_1, B_1^\top$	$A_0, A_1, B_0^\top, B_1^\top$	4	$C_{3,0}$	$A_3, B_0^\top$	$A_0, A_1, B_0^\top, B_1^\top$	3
$C_{1,2}$	$A_1, B_2^\top$	$A_0, A_1, B_2^\top, B_3^\top$	5	$C_{3,1}$	$A_3, B_1^\top$	$A_2, A_3, B_0^\top, B_1^\top$	3
$C_{1,3}$	$A_1, B_3^\top$	$A_0, A_1, B_2^\top, B_3^\top$	5	$C_{2,1}$	$A_2, B_1^\top$	$A_2, A_3, B_0^\top, B_1^\top$	3
$C_{2,0}$	$A_2, B_0^\top$	$A_0, A_1, B_2^\top, B_3^\top$	7	$C_{2,2}$	$A_2, B_2^\top$	$A_2, A_3, B_0^\top, B_1^\top$	4
$C_{2,1}$	$A_2, B_1^\top$	$A_2, A_3, B_0^\top, B_1^\top$	7	$C_{3,2}$	$A_3, B_2^\top$	$A_2, A_3, B_2^\top, B_3^\top$	4
$C_{2,2}$	$A_2, B_2^\top$	$A_2, A_3, B_0^\top, B_1^\top$	8	$C_{3,3}$	$A_3, B_3^\top$	$A_2, A_3, B_2^\top, B_3^\top$	4
$C_{2,3}$	$A_2, B_3^\top$	$A_2, A_3, B_2^\top, B_3^\top$	8	$C_{2,3}$	$A_2, B_3^\top$	$A_2, A_3, B_2^\top, B_3^\top$	4
$C_{3,0}$	$A_3, B_0^\top$	$A_2, A_3, B_0^\top, B_1^\top$	9	$C_{1,3}$	$A_1, B_3^\top$	$A_0, A_1, B_2^\top, B_3^\top$	5
$C_{3,1}$	$A_3, B_1^\top$	$A_2, A_3, B_0^\top, B_1^\top$	9	$C_{1,2}$	$A_1, B_2^\top$	$A_0, A_1, B_2^\top, B_3^\top$	5
$C_{3,2}$	$A_3, B_2^\top$	$A_2, A_3, B_2^\top, B_3^\top$	10	$C_{0,2}$	$A_0, B_2^\top$	$A_0, A_1, B_2^\top, B_3^\top$	5
$C_{3,3}$	$A_3, B_3^\top$	$A_2, A_3, B_2^\top, B_3^\top$	10	$C_{0,3}$	$A_0, B_3^\top$	$A_0, A_1, B_2^\top, B_3^\top$	5

1.2 Objectives

The aim of this thesis was to conduct a study about the algorithms present in [1]. This article describes how can a Space-Filling Curve (SFC) be used to define the scanning order of a given matrix. This order exploits cache spatial and temporal locality,therefore transforming naive implementations of matrix operations into cache-oblivious algorithms able to compete with the most sophisticated BLAS implementations. During the study of these algorithms, we developed an alternative approach, called XOR-Hilbert. This algorithm proven to be competitive against FUR-Hilbert. The main goal of this thesis is to provide a detailed description of this algorithm, its usage, implementation, and also compare the pros and cons of this approach against the approaches developed by [1].

1.3 Document Structure

The initial part of this document, Chapter 2, will start by providing an introduction to Cache-Oblivious algorithms. This chapter will also describe the Ideal Cache model, and provide some examples on how does the scanning order of matrix affects the total amount of Cache-Misses. The analysis of these examples make use of the Ideal Cache model. Finally, different well-known methods to compute the Hilbert Space-Filling Curve will be presented. Chapter 3 will describe three new approaches to compute an Hilbert Space-Filling Curve in linear time. The first of these approaches is called XOR-Hilbert. The remaining two approaches describe a memory optimized version of XOR-Hilbert, and version of the FUR-Hilbert that makes use of XOR-Hilbert in its inner-most loop. Chapter 3 describes the implementation details of each approach, including a brief description on how to transform the computation of an Hilbert Space-Filling Curve into a Cache-Oblivious Nested For-Loop. The run-time and memory analysis of each approach is also presented in this chapter. Chapter 5 presents a comparison between the run-time and cache miss-rate of three different applications, making use of XOR-Hilbert, FUR-Hilbert and the iterative version of the Lindenmayer-System. Chapter 6 concludes this document with some final remarks and suggestions for future work.

2 Background

Contents

2.1 Cache-Oblivious Algorithms

The concept of a cache-oblivious algorithm, revolves around the idea of implementing an algorithm that maximizes the hit-rate of any given cache without previous knowledge of its attributes like level hierarchy, line size and bandwidth. For an algorithm to be cache-oblivious the number of misses must be optimal in an asymptotic sense. High performing algorithms leverage the characteristics of the cpu-cache to obtain a better performance in a absolute sense. These algorithms are said to be cache-aware, however their cache complexity, i.e. number of cache-misses, is exactly the same as the one hold by their cache-oblivious counterpart. Thus, the amount of cache-misses held by these two families of algorithms will only differ by a constant factor. The main advantage of writing cache-oblivious algorithms is its portability.

2.1.1 Ideal-Cache Model

In order to understand the concept of cache obliviousness it is helpful first to be acquainted with the ideal-cache model. Due to the simplicity of this model, one can use this abstraction to simplify the complexity analysis of a given cache-oblivious algorithm.

Definition 2.1. Ideal-Cache Model as described by *Frigo et al.* [3]:
*This model consists of a computer with a two-level memory hierarchy, a Z word ideal cache, and a arbitrarily large main-memory. Main memory is partitioned in memory blocks of size L. Each one of the Z lines that compose this cache has size equal to L. It is assumed our cache is **tall**, $Z = \Omega(L^2)$, meaning the number of cache lines is proportional to L^2. Our cache is assumed to be **fully associative**, allowing a given block to be allocated in any cache line. **Cache-hits** occur when the referenced word is in a line present in cache. **Cache-misses** occur when the referenced word is not in any line present in cache. Whenever a cache-miss occurs the memory block containing the referenced word is fetched from main memory to cache; If our cache is full when a line is fetched from main memory, one line must be evicted from cache. The line to be evicted is assumed to be the one from all Z lines that will be accessed furthest in the future.*

2.1.2 Cache-Oblivious Matrix Transposition

For better understanding on how the analysis of a cache-oblivious algorithm is done, we will perform a simple analysis of a cache-oblivious algorithm to compute the transpose of a given matrix A and store it in another matrix, T, using row-major layout. This example was presented by [3], in this case we will only consider the analysis of the cache misses related with the input matrix, since the analysis of the output is analogous to this one:

Example 2.2. Cache-Oblivious Matrix Transposition:

- *Let A, be a $m \times n$ matrix. Due to the properties of the transpose, the transposed of A, T, will have dimensions $n \times m$;*
- *Let α, be the smallest number that allows $\max(m', n') \leq \alpha L$, where m' and n' are the dimensions of the two sub-matrices of A. In another words, both of this matrices fit completely in cache;*
- *Finally lets assume $m \leq \alpha L < n$, meaning no row of A completely fits in the cache, the case where $n \leq \alpha L < m$ is analyzed analogously;*
- *A divide-and-conquer is performed. At a certain point in time $\alpha L/2 \leq n \leq \alpha L$, since $\max(m, n) \leq \alpha L$ the whole problem fits in cache;*
- *The chosen layout was row-major order, meaning all entries of the input matrix are laid out in contiguous memory. Thus it will be needed to fetch $n.m/L$ blocks from main memory;*
- *In the beginning of the analysis the cache is assumed to be empty, thus a constant number of block fetches $O(1)$, containing data related with the first entries of the input matrix, will be performed.*

Thus concluding that the analysis of the input matrix incurs in at most $O(1+n.m/L)$ cache-misses, when the whole problem fits in cache.

2.2 Layout and Representation of Matrices

A matrix is an abstract mathematical construct, composed by $n \in \mathbb{N}$ rows and $m \in \mathbb{N}$ columns. Each of its $n.m$ entries contains a value, typically a number. In order to efficiently work with matrices in a computer science context it is important to understand how these structures are stored in memory. In other words, what memory layout was used to store a given matrix. The memory layout of a given structure is defined by the programming language that was used to implement it. There is no defined standard and some programming languages are able to store a matrix using several different layouts. Three of the most common types of layouts are the row-major order [4], used in C, C++ and Pascal. Column-Major Order [4], used in FORTRAN, Matlab and Julia. And lastly the Iliffe Vector [5], used in Java, .Net and Scala. We will now provide several examples on how a matrix is stored in memory, according to each of the previously referenced layouts. A theoretical analysis of the cache behaviour will also be performed for different types of matrix scans. This analysis will make use of the Ideal-Cache model, described in Section 2.1.1.

For the following examples assume that M is a generic $n \times m$ matrix, i.e. a matrix with n rows and m columns, where $n, m \in \mathbb{N}$. Each entry, or cell, $M_{i,j}$ will be represented using programming notation, $M[i][j]$. The variables i and j work as entry indexes, and are bounded by $0 \leq i < n, i \in \mathbb{N}_0$ and

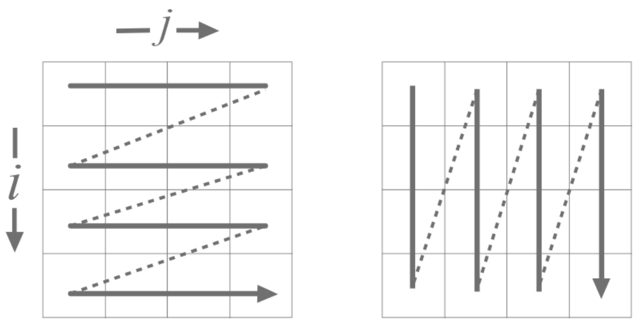


Figure 2.1: Row-major order (on the left) and column-major order (on the right).

$0 \leq j < m, j \in \mathbb{N}_0$. The value located at the top left-most entry of M can be referenced by $M[0][0]$, while the bottom right-most entry can be referenced by $M[n-1][m-1]$.

2.2.1 Row-Major Order

This layout is depicted in Figure 2.1 (left-side figure). All elements of a given row, contained by matrix M, are stored in adjacent memory addresses. Each entry is stored in a sequential fashion, ordered by index j. Thus the memory addresses that come immediately before $M[i][j]$, $(\forall j \geq 1)$, contain information regarding $M[i][j-1]$. Conversely the memory address that succeeds the memory bounds of $M[i][j]$, $(\forall j < m-1)$, will contain information regarding $M[i][j+1]$. In its turn all the n rows of M will also be allocated in a sequential manner, ordered by index i, from 0 to $n-1$. Implying that the memory address that precedes the memory bounds of $M[i][0]$, $(\forall i \geq 1)$, will contain information regarding $M[i-1][m-1]$. While the memory address that succeeds entry $M[i][m-1]$, $(\forall i < n-1)$, will contain information associated with $M[i+1][0]$.

2.2.2 Column-Major Order

The logic behind this layout is similar to row-major order, and can be thought as the transposed counterpart of the previous layout, as depicted on the right-side of Figure 2.1. In column-major order all elements of a given column, contained by the same matrix M, are stored in adjacent memory locations. These elements are ordered from 0 to m according to the value of index i. In this case the memory addresses that immediately precedes $M[i][j]$, $(\forall i \geq 1)$, contain information regarding $M[i-1][j]$. The information regarding $M[i+1][j]$ is found by the memory address that succeeds the memory bounds of $M[i][j]$, $(\forall i < n-1)$. In this model all m columns of M are allocated in adjacent memory slots, now ordered by index j from 0 to $m-1$. Thus the address that precedes the memory bounds of $M[0][j]$, $(\forall j \geq 1)$, will contain information regarding $M[n-1][j-1]$. While the memory address that succeeds entry $M[n-1][j]$, $(\forall j < m-1)$, will contain information associated with $M[0][j+1]$.

Algorithm 2.1: Row-Major Scan

```
Input: $n$
begin
    for $i := 0; i < n; i := i + 1$ do
        for $j := 0; j < n; j := j + 1$ do
            $M[i][j] := M[i][j];$
```

Algorithm 2.2: Column-Major Scan

```
Input: $n$
begin
    for $j := 0; j < n; j := j + 1$ do
        for $i := 0; i < n; i := i + 1$ do
            $M[i][j] := M[i][j];$
```

2.2.3 Iliffe Vector

This layout, created by Iliffe in 1961 [5], is a variant of the row-major order. Similarly with what happens in row-major order, all elements within a given row of matrix M, are allocated in adjacent memory slots and ordered in the same way as the first layout. However this model was design to deal with memory fragmentation while allocating memory during the run-time of a program. In order to overcome this problem the approach taken was to allocate each row in the memory slot that would best reduce memory fragmentation. This renders the task of finding the offset of a given entry that is not located in the first row unfeasible.

The usage of a different layout should not lead to any differences in performance, with the exception of Iliffe Vectors that were design to minimize memory fragmentation. Nonetheless the programmer must adapt his code to the memory layout employed by the programming language in use. Otherwise the cache hit-rate will be sub-optimal, which in its turn causes the global run-time of the program to be sub-optimal. This can be demonstrated by Algorithms 2.1 and 2.2. Both algorithms perform a complete scan of matrix M, visiting each entry only once. The total amount of operations performed by each scan is exactly the same.

Eric D. Demaine [6] describes and proves a theorem that guarantees that scanning N elements stored in a contiguous segment of memory incurs at most in $\lceil N/L \rceil + 1$ cache misses. Assume that M, our previously defined $n \times m$ matrix, is allocated in memory using row-major layout and scanned using the Row-Major Scan described in Algorithm 2.1. Since the k^{th} entry of M is adjacent in memory to the $k - 1^{th}$ and $k + 1^{th}$ entry, the complete scan of this matrix will incur in $\lceil n.m/L \rceil + 1$ cache-misses, which is optimal. Lets instead assume that the scanning algorithm used to traverse M was Algorithm 2.2. M is still allocated in memory using row-major layout. Two different cases will be now analyzed, using the ideal-cache model. The first presents a cache that will incur in the same amount of misses as the previous, while the second case presents a cache that will generate a far greater amount of blocks fetched:

Example 2.3. *Assume that $L.Z \geq n.m$, and $L \leq m$. At the beginning of Algorithm 2.2 our cache is completely empty, thus in order to scan the first n elements of M, n blocks must be fetched from main* *memory to cache. Since $L \times Z \geq n \times m$ the whole problem fits in cache, implying no cache line will be replaced during the run-time of Algorithm 2.2. Thus no further cache-miss will occur until all the other $L - 1$ entries, present in each of the previously n fetched memory blocks, are scanned. This leads to a number of m/L misses per scanned row, which in its turn implies a total number of $n.m/L$ cache-misses. This amount of cache-misses can be increased by 2 if the first and last entries to be scanned are allocated at the end and beginning, of their respective memory block.*

Example 2.4. *Assume that $L \leq m$ and that $Z \leq n$. Following the ideal-cache model, our cache must be tall and the replacement method is optimal. Similarly to the previous example we start with an empty cache. If one employs Algorithm 2.1 to scan this matrix the number of cache-misses will be the same as in Example 2.3, since all matrix lines are contiguous in memory. However, if this matrix is scanned by Algorithm 2.2 all adjacent matrix entries will be found in different memory blocks since $L \leq m$. Furthermore, $Z \leq n$ and the replacement policy ensure that after scanning a complete column only one of the previously fetched memory blocks will remain in cache. Therefor Algorithm 2.2 will result in $m + (m-1)(n-1)$ cache-misses. In a real world scenario this situation would actually result in mn misses, since the replacement policy in most modern caches is the Least-Recently-Used (LRU).*

2.3 Space-Filling Curves

In Mathematical Analysis, a space-filling curve is a 1-dimensional continuous curve, whose properties allow it to contain all points in a unit-square within its range. It is proven this family of curves can also fill in the 3-dimensional unit-cube, as well as hyper-geometrical volumes. Two of the most well known SFC's are the Peano and Hilbert curves. Both of these belong to a smaller family of curves, called FASS curves. FASS is an acronym for approximate space-filling, self-avoiding, self-similar and simple, as described in [7] . A curve is considered to belong to the FASS family if:

- **Approximate Space-Filling**, if the curve passes within a small distance from all points of a given two-dimensional figure, such as a square, which surrounds the curve. This distance can be arbitrarily reduced by carrying on the recursive construction to a greater level of recursion;
- **Self-Avoiding**, if the segments of the curve do not touch or intersect;
- **Self-Similar**, if the construction of the curve can be done by applying a recursive set of rules to connect components;
- **Simple**, if the curve can be draw by a single stroke of a pen, without lifting the pen nor drawing any segment more than once.

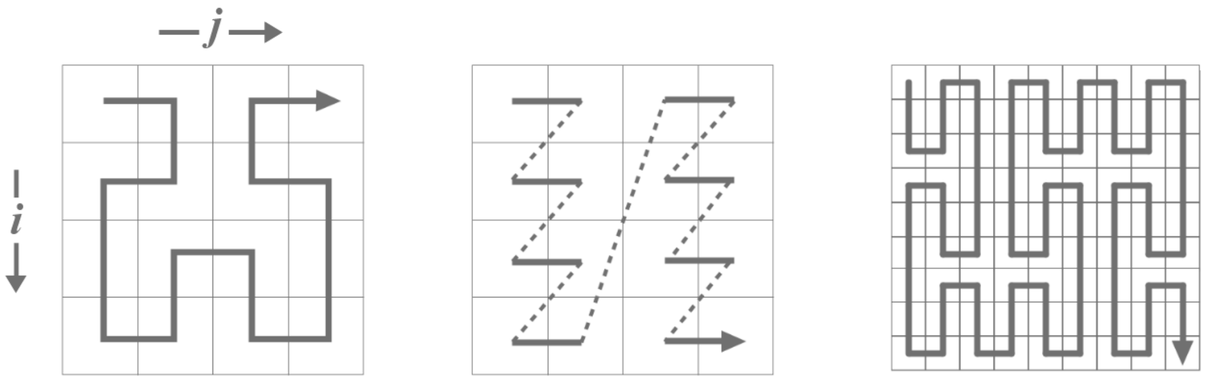


Figure 2.2: Second iterations of Hilbert curve (on the left), z-order curve (on the middle), and Peano curve (on the right).

These properties provide enough evidence to reach the conclusion that a one-to-one mapping exists between the points of a FASS curve and the points present in a unit-square, if represented by a grid. In other words, there is a bijective function $f(i) \rightarrow (x, y)$, where $i \in \mathbb{N}_0$ and $(x, y) \in (\mathbb{N}_0, \mathbb{N}_0)$. Index i represents the i^{th} point of the FASS curve, and pair (x, y) represents a point of the unit-square. It is important to note that for this one-to-one mapping exist a SFC does not need to belong to the FASS curve family. In fact one of the most used SFC's in computer science is the Lebesgue curve, also known as z-order curve. As it can be seen in Figure 2.2, this curve does not comply with the simplicity requirement of FASS curves. Each time we step into a different quadrant or sub-quadrant of the unit-square, the next point to be filled by this curve will not be a point adjacent to the current one, as explained by [8]. If we visualize the unit-square as a grid, the last point of a given quadrant and the first point of the the quadrant that succeeds it has a Manhattan distance equal to a power of 2, the exponent will depend on the dimensions of the current quadrant. As previously stated a cache-oblivious traversal has to exploit spatial locality, which does not happen when traversing the points of grid using a z-order traversal. This lead us to consider the Lebesgue curve as unsuitable to perform a cache-oblivious matrix traversal. Our study will be focused in FASS curves, more specifically in the Hilbert curve, since this thesis is built upon Bohm et al. [1] work.

2.3.1 Hilbert Curve Properties and Restrictions

Böhm et al. [1] defines a mapping function similar to the one previously defined in the beginning of this section. This function is called the Hilbert Inverse $H^{-1}(x)$, mapping each point of an Hilbert curve to a grid entry. Being the Hilbert curve part of the FASS family, it is guaranteed that by enumerating any two successive arguments of $H^{-1}(x)$ the returned grid entries will be horizontally or vertically adjacent. Thus preserving locality between adjacent entries of this grid. This property can also be demonstrated without recurring to FASS curves properties since:

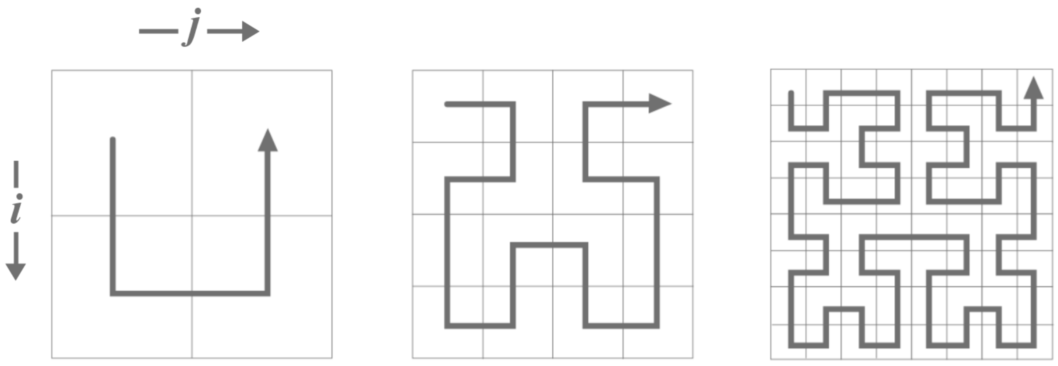


Figure 2.3: First three iterations of a traditional Hilbert curve.

$$H^{-1}(x+1) = \begin{cases} H^{-1}(x) \pm (1,0) \\ H^{-1}(x) \pm (0,1) \end{cases} \tag{2.1}$$

Although this curve seems ideal in terms of locality preservation it still presents some drawbacks. Computing this curve can be computationally hard, and in order to fill a grid with this curve, it is required that this grid has dimensions equal to $n \times n$, where $n = 2^l$ and $l \in \mathbb{N}$ represents the l^{th} level of recursion, or resolution, of this curve. However the impossibility of generalizing a SFC to fill a grid with arbitrary side lengths is common to all space-filling curves. A pure Peano curve will fill squared grids where the side length is a power of 3, and a pure z-order curve will only fill squared grids with side length equal to a power of 2. Even though the restrictions presented by Hilbert and Peano curve seem identical, the Hilbert curve presents greater coherence [9], i.e. locality, than Peano.

2.4 Computing the Hilbert Curve

This section will describe several well-known methods to compute a complete 2-dimensional Hilbert curve, along with the analysis of time and memory complexity of each of these approaches. Most of the well-known approaches that compute a complete Hilbert curve are decoders that convert the h^{th} entry into a pair of 2d-coordinates, (x, y). This decoders manipulate the number of bits present in h in order to return a pair of coordinates, requiring at least $O(\log(N))$ operations per decoding. Obtaining all coordinates incur in $O(N\log(N))$ operations. Example of these types of decoders are the Mealy-DFA as defined in [10], or other approaches that scan the bits of current iteration value h, as [11]. Other approaches can compute this SFC through recursive calls, which present an overhead in the form of jumps within the function call stack and cannot be optimized by most of the compilers, the most well-known being the Lindenmayer-System [12].Our study will be mainly focused on the iterative approaches presented by *Böhm et al.* [1].

2.4.1 Lindenmayer-System (Recursive)

These systems are built upon an appropriate Context-Free Grammar [13], composed by terminal symbols, non-terminal symbols, and production rules. This approach receives as input the desired level of recursion/iteration of a given Space-Filling Curve, and returns either a draw of the computed curve, or a string describing this path.

2.4.1.A Turtle Notation

One of the traditional methods used to explain how does a Lindenmayer-System works is to resort to the Turtle Notation, described in [14]. Simply put, one can imagine the output of a given Lindenmayer-System as sequence of commands to be followed by an imaginary turtle. This turtle acts as a pawn within the grid we want to traverse, knowing only to which direction it is ”looking”. Starting at a given entry of this grid, our turtle will consume two type of commands: `Move-Forward` n units or `Rotate` α degrees. The actions this turtle must perform are encoded within each terminal symbol of this system. The terminal symbols present in this grammar are: the Forward-Step ($\triangleright$), which represents a `Move-Forward` command by 1 unit; the Look-Left ($\ominus$), representing a `Rotate` by 90 degrees command; and finally the Look-Right ($\oplus$), which represents a `Rotate` by -90 degrees command.

2.4.1.B Algorithm Overview

The Lindenmayer-System used to compute an arbitrarily large iteration of an HSFC is defined in [1]. We will now describe this approach by making use of the previously defined Turtle Notation, thus providing a more intuitive explanation.

Let the set of non-terminal symbols be composed by A, and B, while the set of terminal symbols is constituted by $\ominus$, $\oplus$, $\triangleright$, and π (not present in conventional Systems). The two production-rules that define this system are:

$$A \rightarrow \ \pi | \ominus B \triangleright \oplus A \triangleright A \oplus \triangleright B \ominus, \tag{2.2}$$

$$B \rightarrow \ \pi | \oplus A \triangleright \ominus B \triangleright B \ominus \triangleright A \oplus . \tag{2.3}$$

The domain of variable d is equal to $\{0, 1, 2, 3\}$. The `move-forward` command, depending on variable

Algorithm 2.3: Rule A

```
Input: $l$
begin
    $(i, j) := (0, 0)$;
    $d := 3$;
    if $l = 0$ then
        process_entry$(i, j)$
    else
        $d := (d + 3) \bmod 4$;
        $B(l - 1)$;
        $i := i + (d - 2) \bmod 2$;
        $j := j + (d - 1) \bmod 2$;
        $d := (d + 1) \bmod 4$;
        $A(l - 1)$;
        $i := i + (d - 2) \bmod 2$;
        $j := j + (d - 1) \bmod 2$;
        $A(l - 1)$;
        $d := (d + 1) \bmod 4$;
        $i := i + (d - 2) \bmod 2$;
        $j := j + (d - 1) \bmod 2$;
        $B(l - 1)$;
        $d := (d + 3) \bmod 4$;
```

Algorithm 2.4: Rule B

```
Input: $l$
begin
    $(i, j) := (0, 0)$;
    $d := 3$;
    if $l = 0$ then
        process_entry$(i, j)$
    else
        $d := (d + 1) \bmod 4$;
        $A(l - 1)$;
        $i := i + (d - 2) \bmod 2$;
        $j := j + (d - 1) \bmod 2$;
        $d := (d + 3) \bmod 4$;
        $B(l - 1)$;
        $i := i + (d - 2) \bmod 2$;
        $j := j + (d - 1) \bmod 2$;
        $B(l - 1)$;
        $d := (d + 3) \bmod 4$;
        $i := i + (d - 2) \bmod 2$;
        $j := j + (d - 1) \bmod 2$;
        $A(l - 1)$;
        $d := (d + 1) \bmod 4$;
```

d value, is encoded by the following statements:

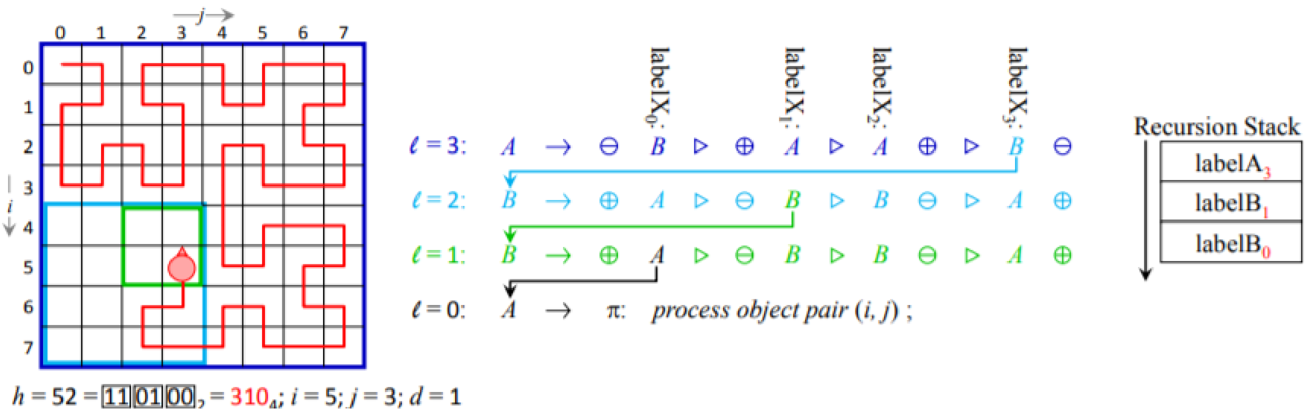
$d = 0 \Leftrightarrow$ *looking right, after next* $\triangleright$ *command* $\Rightarrow j := j + 1$,

$d = 1 \Leftrightarrow$ *looking up, after next* $\triangleright$ *command* $\Rightarrow i := i + 1$,

$d = 2 \Leftrightarrow$ *looking left, after next* $\triangleright$ *command* $\Rightarrow j := j - 1$,

$d = 3 \Leftrightarrow$ *looking down, after next* $\triangleright$ *command* $\Rightarrow i := i - 1$.

Terminal symbol π can be seen as asking our turtle its current coordinates (i, j). Terminal symbols $\ominus$ and $\oplus$ are encoded through the mathematical modulo operator. An $\ominus$ command is encoded as statement $d := (d + 1) \bmod 4$ while $\oplus$ can be encoded by $d := (d + 3) \bmod 4$. Traditionally the starting production rule, or axiom, of a Lindenmayer-System designed to compute an Hilbert Curve, is either non-terminal symbol A with initial direction $d = 3$, or B with initial direction $d = 2$. Thus generating either an clockwise, or anticlockwise oriented curve, respectively. Since a proper enconding is known for every production rule and terminal symbol, it is quite simple to translate this Lindenmayer-System to code, as depicted by Algorithms 2.3 and 2.4 that correspond to production rule A and B, respectively.



$h = 52 = \boxed{11}\boxed{01}\boxed{00}_2 = 310_4$; $i = 5$; $j = 3$; $d = 1$
$\ell = 3$: $A \rightarrow \ominus\ B \triangleright \oplus\ A \triangleright\ A \oplus \triangleright\ B \ominus$
$\ell = 2$: $B \rightarrow \oplus\ A \triangleright \ominus\ B \triangleright\ B \ominus \triangleright\ A \oplus$
$\ell = 1$: $B \rightarrow \oplus\ A \triangleright \ominus\ B \triangleright\ B \ominus \triangleright\ A \oplus$
$\ell = 0$: $A \rightarrow \pi$: process object pair (i, j) ;
labelX$_0$:
labelX$_1$:
labelX$_2$:
labelX$_3$:
Recursion Stack
labelA$_3$
labelB$_1$
labelB$_0$

Figure 2.4: Recursive generation of (i, j)-pairs following the Hilbert curve. This figure was taken from [1].

2.4.1.C Run Time and Memory Considerations

All terminal symbols, $\oplus, \ominus, \triangleright$ and π, are encoded using a constant number of operations. This leads to the assumption that exists a constant overhead, $O(1)$, per element computed. The recursive nature of this algorithm implies a logarithmic overhead in the recursion stack. In the worst case scenario, after every 4^k loop iterations the stack-pointer has to move up or down at least k positions, where $k \in [1, \log N]$. Memory wise, although this code seems to allocate a constant amount of memory it actually needs to store the function calls in at least l stack positions. Thus requiring $O(l)$, or $O(\log N)$ memory slots in the stack. This may also lead to a stack overflow error. One last drawback of the recursive nature of this algorithm is that compilers do not optimize code between functions, as stated in [1].

2.4.2 Lindenmayer-System (Iterative)

In order to overcome the drawbacks of the approach presented in Section 2.4.1, *Böhm et al.* [1] present us an alternative method to compute an HSFC in a completely iterative fashion. The approach taken consists in emulating the behaviour of the function call stack of the recursive Lindenmayer-System, based on the observation that the bit-pairs used to represent h in *4-dic*, contain the same values as the function labels present in the recursion stack, as depicted by Figure 2.4.

2.4.2.A Algorithm Overview

LabelA_x and LabelB_x, present in the previous figure, represent the x^{th} non-terminal symbol expanded in production rule A or B, respectively. Lemmas 2.5 to 2.7 define how to obtain the action code, i.e. label number, in order to iteration step h. These lemmas are proved and provided in [1].

Lemma 2.5. The number $\texttt{pop}(l)$ of processed object pairs (i, j) starting from the incarnation at level l is $\texttt{forw}(l) = l^4$.

Lemma 2.6. The incarnation overall increases h by $\texttt{forw}(l) = l^4 - 1$. Where $\texttt{forw}(l)$ is the number of forward-steps present in l^{th} iteration of an Hilbert Curve.

Lemma 2.7. For each Hilbert value h generated at label$\mathbf{X}_k$, the value $a := \lfloor h/4^{l-1} \rfloor \ mod\ 4$ equals the label number k. Label$\mathbf{X}_k$ represents the k^{th} non-terminal symbol that is being expanded in A or B.

Since the label numbers are known, in order to emulate the current state of the recursion stack one just needs to know to which production rule, A or B, does the label number refers to. The following lemmas were also proven in [1].

Lemma 2.8. The direction code d is the same at the beginning and at the end of a production rule.

Lemma 2.9. 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.

From Lemma 2.7 and Lemma 2.9, it follows that the parity of direction d and the label tag is sufficient to determine if the currently expanded rule is either A or B. Thus resulting in Lemma 2.10:

Lemma 2.10. The parity of d combined with the position (e.g. the label tag) determines if we are in grammar rule A or B.

It is now known that only certain direction values, d, are possible for a given label$\mathbf{X_k}$. Making it feasible to calculate Tables 2.1 and 2.2, which state how the value of variable d is altered in order to its current value, production rule, and label tag. Symbol ”-”, means that d remains unchanged. These lemmas and observations result in Algorithm 12. Where Lemma 2.5 defines the number of loop iterations, at line 4. Action code a is computed as stated in Lemma 2.7, at line 7. The action code transition, as depicted by Tables 2.1 and 2.2 can be found in lines 9 and 12. And finally, the current coordinates within our grid are computed using the same method as in Section 2.4.1.B.

One last detail that must be presented is the computation of level variable l. The value held by this variable is encoded by number of least-significant 00_{bin} bit-pairs of h. Variable l is calculated using a bit-hack to compute the logarithm of base 2 in constant time, as seen in [15].

2.4.2.B Run-Time and Memory Analysis

In contrast to the approach used in Section 2.4.1, this solution only allocates one function pointer in the stack due to its iterative nature. Thus eliminating stack jumps during the execution of the algorithm. The while-loop of this algorithm contains only elementary operations, and the number of operations is constant (24). Thus this algorithm presents a constant overhead, $O(1)$, per loop iteration. All the variables present in this algorithm can be represented using integers, commonly implemented in C using 32 bits. Implying this algorithm requires a constant, $O(1)$, amount of memory.

Table 2.1: Transition table for variable d for odd l values.
$\ominus/\oplus$ before $\triangleright$ (left table), and $\ominus/\oplus$ after $\triangleright$ (right table).

$d_{old} = 0$	$a = 1$	$a = 2$	$a = 3$
0	-	-	3
1	-	-	2
2	-	-	1
3	-	-	0

$d_{old} = 0$	$a = 1$	$a = 2$	$a = 3$
0	1	-	-
1	0	-	-
2	3	-	-
3	2	-	-

Table 2.2: Transition table for variable d for even l values.
$\ominus/\oplus$ before $\triangleright$ (left table), and $\ominus/\oplus$ after $\triangleright$ (right table).

$d_{old} = 0$	$a = 1$	$a = 2$	$a = 3$
0	3	3	-
1	2	2	-
2	1	1	-
3	0	0	-

$d_{old} = 0$	$a = 1$	$a = 2$	$a = 3$
0	1	1	-
1	0	0	-
2	3	3	-
3	2	2	-

Algorithm 2.5: Iterative Lindenmayer-System

```
Input: $n$
begin
    $(i, j) := (0, 0)$;
    $h := 0$;
    $d = 3$;
    while $\underline{h < n^2}$ do
        process_entry$(i, j)$;
        $h := h + 1$;
        $l := \log_2 \lfloor \frac{1}{2}(h \; \&_{bitw}(-h)) \rfloor$;
        $a := \lfloor h/4^{l-1} \rfloor$;
        $d := d \; \mathbf{xor_{bitw}}(11_2.(\mathbf{isOdd}(l-1) \; \mathbf{xor_{bitw}} \; a = 3))$;
        $i \; := i + (d-2) \bmod 2$;
        $j \; := j + (d-1) \bmod 2$;
        $d \; := d \; \mathbf{xor_{bitw}}(\mathbf{isOdd}(l-1) \; \mathbf{xor_{bitw}} \; a = 1)$;
```

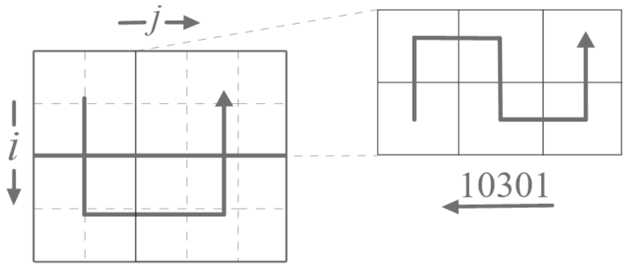


Figure 2.5: Example on how to apply a Nano-Program to a sub-grid.

2.4.3 FUR-Hilbert

Even though the previous approaches compute a complete Hilbert Curve with a reasonably fast run-time, the main drawback of these approaches is that it's impossible to scan any matrix that is not a square matrix with row/column size equal to a power of 2. *Böhm et al.* [1] present us a different approach that makes use of small pre-processed FASS curves called Nano-Programs. The curves encoded by these Nano-Program can fill grids with size $r \times s = 2 \times 2, 2 \times 3, 2 \times 4, 3 \times 4, 4 \times 4$, as well as single loops $1 \times \{1, 2, 3, 4\}$, and empty loops $0 \times \{1, 2, 3, 4\}$. Each Nano-Program is represented by a sequence of pair of bits, where each pair represents a direction in Turtle Notation [16]. The domain of possible directions values is equal to $\{0, 1, 2, 3\}$, which is the same as the one presented in Section 2.4.1.B. Converting a given direction to a coordinate-pair of our grid follows the same formula as Section 2.4.1.B. Nano-Programs are read from right to left, i.e. the first direction to be processed is found in the least-significant pair of bits of the variable that represents a given Nano-Program, while the last direction to be processed can be found in the most-significant pair of bits of this variable. In order to obtain succeeding directions contained by a given Nano-Program, one must apply a bit-wise shift-left ($>>$) operation by 2, or the equivalent integer division by 4 operation to this sub-path, followed by a modulus 4 operation.

2.4.3.A Algorithm Overview

However, the path contained by these Nano-Programs are too small to fill an arbitrarily large rectangular grid. *Böhm et al.* [1] proved that is possible to tesselate a grid with dimensions $n \times m$ with a given amount of sub-grids with size $r \times s = \{2, 3, 4\} \times \{2, 3, 4\}$, assuming $\lfloor \log_2(n) \rfloor = \lfloor \log_2(m) \rfloor$. If one pictures this grid thinking of each sub-grid that compose it as a single grid entry, then the result will be a grid with dimension $t \times t$, where $t = \lfloor \log_2(n) \rfloor = \lfloor \log_2(m) \rfloor$. Implying this new grid can be traversed by an HSFC with level equal to $\lfloor \log_2(t) \rfloor$. In their implementation, this HSFC is computed through the iterative Lindenmayer-System approach, as it can be seen in Algorithm 2.6. After the inter-sub-grid traversal, one simply needs to call the correct Nano-Program in order to perform the intra-sub-grid traversal.

2.4.3.B Odd-sized grid

The FUR-Hilbert algorithm deals with odd-sized sub-grids in the following fashion. If exactly one of the dimensions of this grid has odd size, i.e. either $n \in \mathcal{O}$ or $m \in \mathcal{O}$, where $\mathcal{O}$ represents the sets of natural numbers that are odd, then the grid is filled with sub-grids with even dimensions sub-grids, except for the sub-grids present in the last row or column, depending whether n or m belongs to $\mathcal{O}$, respectively. On the other hand, if both dimensions of this grid are odd-sized then last column to be traversed will be filled with 3×2 sub-grids and a single 3×3 sub-grid, either at the top or bottom of this column. Through bit-wise operations the algorithm decides where to place the 3×2 sub-grids, present in middle of the this grid, as depicted by Figure 2.5.

2.4.3.C Severely Asymmetric Grid

Given that the maximum grid size a Nano-Program can fill is a sub-grid of 4×4, it is impossible to connect all sub-grids that compose a grid whose dimensions respect any of the following inequalities, $\lfloor \log_2(n) \rfloor < \lfloor \log_2(m) \rfloor$, or $\lfloor \log_2(n) \rfloor > \lfloor \log_2(m) \rfloor$, with a single Hilbert Space-Filling Curve. Assuming $\lfloor \log_2(n) \rfloor < \lfloor \log_2(m) \rfloor$, *Böhm et al.* [1] overcome this problem by computing $m' = \lceil m/2^{\lfloor \log_2 n \rfloor} \rceil$ independent curves side by side. The first curve to be computed has width $m - \lfloor \log_2 n \rfloor \dot{(} m' - 1)$. The remaining $m' - 1$ curves will have width equal to $\lfloor \log_2 n \rfloor$. All these curves are anticlockwise (initial direction equal to 2). If $\lfloor \log_2(n) \rfloor > \lfloor \log_2(m) \rfloor$ would be verified, the computation of this curve would be analogous to the previous one, the starting direction would be $d = 3$.

2.4.3.D Run-Time and Memory Complexity

Similarly to what happens in the previous approach, Section 2.4.2, this algorithm as linear time complexity, $O(N)$. However this code is significantly faster than the iterative Lindenmayer-System approach, since the processing time of each entry is amortized by the use of Nano-Programs, thus reducing considerably the run-time of this algorithm in comparison with the previously stated approach. The memory complexity of FUR-Hilbert is constant,$O(1)$, albeit large, since its implementation encodes each Nano-Program in `unsigned long long` C coding language variables. As stated in [1] this algorithm requires 100 Nano-Programs to be coded, and the C Standard (ISO C99) [17] ensures that this type of variable has at least 64 bits, resulting in at least 800 bytes of memory used to encode all Nano-Programs.

Algorithm 2.6: FUR-Hilbert

```
Input: $(I, J)$, $(i_{max}, j_{max})$
begin
    $d := 0$;
    $t := \lfloor \log_2 n \rfloor$;
    while $h < 2^{2t-2}$ do
        $r := \lfloor \frac{(I+1)n}{t} \rfloor - \frac{I \cdot n}{t}$;
        $d := \lfloor \frac{(J+1)m}{t} \rfloor - \frac{J \cdot m}{t}$;
        $P :=$ nanoPrograms$[r][s][d]$;
        while $P \neq 0$ do
            processObjectPair$(i, j)$;
            $c := P \bmod 4$;
            $P := \lfloor P/4 \rfloor$;
            $j := j + ((c-1)) \bmod 2$;
            $i := i - ((2-c)) \bmod 2$;
        $h := h + 1$;
        $l := \log_2 \lfloor \frac{1}{2}(h \,\&_{bitw} (-h)) \rfloor$;
        $a := \lfloor h/4^{l-1} \rfloor$;
        $d := d\ \mathbf{xor_{bitw}}(11_2.(\mathbf{isOdd}(l-1)\ \mathbf{xor_{bitw}}\ a = 3))$;
        $i\ := i + (d-2) \bmod 2$;
        $j\ := j + (d-1) \bmod 2$;
        $J := J + ((d-1) \bmod 2)$;
        $I := I + ((d-2) \bmod 2)$;
        $d\ := d\ \mathbf{xor_{bitw}}(\mathbf{isOdd}(l-1)\ \mathbf{xor_{bitw}}\ a = 1)$;
```

3 Alternative Approaches

Contents

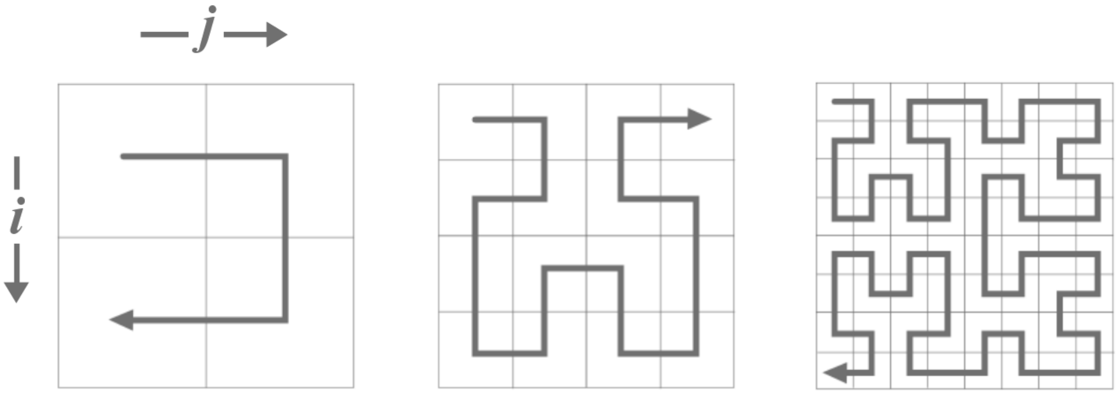


Figure 3.1: First three iterations of the Hilbert curve, built in a incremental manner.

3.1 XOR-Hilbert

Through the study of repeated patterns that compose an Hilbert curve, we found out that is possible to obtain the next iteration of an HSFC through a sequence of bit-wise exclusive-or operations and string concatenations. The pseudo-code of this algorithm is presented in Algorithm 3.1. This algorithm produces the next iteration of a given HSFC in a incremental fashion, as depicted by Figure 3.1. Thus, the first quadrant of the next iteration is always equal to the curve previously held by variable p, which contains the directions that encode the path for iteration l. The remaining quadrants are computed by applying an exclusive-or operation over all elements of p. Functions $\mathbf{xor_x}$ represent this sequence of bit-wise operations, where x represents the value by which we "xor" every element of its argument. In order to compute the whole curve one just needs to link all quadrants with variables s_1, s_2, and s_3, during the process of concatenation. The values of these variables are calculated based on the parity of the iteration number. If one runs this algorithm, starting from the first iteration of an HSFC, it is possible to obtain the L^{th} iteration of this curve in $L-1$ iterations of the for-loop present at line 5 of Algorithm 3.1. After completing this cycle the curve, held by p, is then processed in a similar fashion to what happens in Section 2.4.1.B.

3.2 Proof of Correctness

In order to achieve the greatest degree of accuracy possible, while comparing different approaches, our algorithm must draw the same curves as the ones produced by the algorithms present in Section 2.4. These methods produce a curve equivalent to the one produced by the recursive version of a Lindenmayer-System that uses B with direction $d = 3$ as axiom (odd iterations), or A with direction $d = 2$ (even iterations). Even though this seems to contradict Lemma 2.9, the only changes to this curve are its starting and ending point, leaving the rest of its structure unaltered. For now we are neglecting the process_curve. It is already established from Section 2.4 that it is possible to process the whole curve in $O(N)$.

In order to prove the statements presented in the previous section, it is needed to understand that our implementation ensures that functions $\mathbf{xor_1}$ and $\mathbf{xor_2}$ are associative and idempotent. Therefore, applying these functions to a curve or each of its elements will produce the same result, and if $\mathbf{xor_x}(a) = b$ then $\mathbf{xor_x}(b) = a$. The following lemmas will be proven using induction over a Lindenmayer-System where production rule B with direction $d = 3$, $B_{d=3}$, is used as axiom. Due to the properties of function $\mathbf{xor_x}$, if a condition holds for axiom $B_{d=3}$ it will also hold for axiom $A_{d=2}$. In order to simplify these proofs, some changes were made to the notation in use. From now on l will represent the current iteration, or resolution, of this curve.

Lemma 3.1. $\forall_{l\geq 1}\ \mathbf{xor_1}(B^l_{d=3}) = A^l_{d=2} \cap \mathbf{xor_1}(B^l_{d=1}) = A^l_{d=0}$.

Proof. If one would expand each of the possible axioms once, A^l and B^l for $l \geq 1$, the following strings would be obtained:

$$A^l \xrightarrow{\text{Expand }A} \ominus B^{l+1} \triangleright \oplus A^{l+1} \triangleright A^{l+1} \oplus \triangleright B^{l+1} \ominus, \tag{3.1}$$

$$B^l \xrightarrow{\text{Expand }B} \oplus A^{l+1} \triangleright \ominus B^{l+1} \triangleright B^{l+1} \ominus \triangleright A^{l+1} \oplus\ . \tag{3.2}$$

Lemma 2.9 states that the direction before and after expanding a non-terminal symbol is the same,

Algorithm 3.1: XOR-Hilbert

```
Input: $L$
begin
    $p := 321$;
    $s1 := 2$;
    $s2 := 3$;
    $s3 := 0$;
    for $l := 1; l < L; l := l + 1$ do
        $p := p \cdot s1 \cdot \mathbf{xor_1}(p) \cdot s2 \cdot \mathbf{xor_1}(p) \cdot s3 \cdot \mathbf{xor_2}(p)$;
        $s_1 := s_1 \wedge 1$;
        $s_2 := s_2 \wedge 1$;
        $s_3 := s_3 \wedge 1$;
    process_curve($p$);
```

thus establishing the following equivalences with Formulas 3.1 and 3.2:

$$A^l_{d=0} \xrightarrow{\text{Expand A}} B^{l+1}_{d=1} \triangleright_{d=1} A^{l+1}_{d=0} \triangleright_{d=0} A^{l+1}_{d=0} \triangleright_{d=3} B^{l+1}_{d=3}, \tag{3.3}$$

$$B^l_{d=1} \xrightarrow{\text{Expand B}} A^{l+1}_{d=0} \triangleright_{d=0} B^{l+1}_{d=1} \triangleright_{d=1} B^{l+1}_{d=1} \triangleright_{d=2} A^{l+1}_{d=2}, \tag{3.4}$$

$$A^l_{d=2} \xrightarrow{\text{Expand A}} B^{l+1}_{d=3} \triangleright_{d=3} A^{l+1}_{d=2} \triangleright_{d=2} A^{l+1}_{d=2} \triangleright_{d=1} B^{l+1}_{d=1}, \tag{3.5}$$

$$B^l_{d=3} \xrightarrow{\text{Expand B}} A^{l+1}_{d=2} \triangleright_{d=2} B^{l+1}_{d=3} \triangleright_{d=3} B^{l+1}_{d=3} \triangleright_{d=0} A^{l+1}_{d=0}. \tag{3.6}$$

Base Case:*The curve generated by applying $\mathbf{xor_1}$ over $B^{L=1}_{d=3}$ after expanding it once, is the same generated by expanding $A^{L=1}_{d=2}$ once. The same is verified for $B^{L=1}_{d=1}$ and $A^{L=1}_{d=0}$. The base case holds.*

$$A_{d=2} \rightarrow\ 3\ 2\ 1_{dec} = 11\ 10\ 01_{bin}, \qquad A_{d=0} \rightarrow\ 1\ 0\ 3_{dec} = 01\ 00\ 11_{bin},$$

$$B_{d=3} \rightarrow\ 2\ 3\ 0_{dec} = 10\ 11\ 00_{bin}, \qquad B_{d=1} \rightarrow\ 0\ 1\ 2_{dec} = 00\ 01\ 10_{bin},$$

$$\mathbf{xor_1}(B_{d=3}) = A_{d=2} \qquad \mathbf{xor_1}(B_{d=1}) = A_{d=0}$$

$$\Longleftrightarrow \qquad \Longleftrightarrow$$

$$10\ 11\ 00_{bin} \wedge\ 01\ 01\ 01_{bin} = 11\ 10\ 01_{bin}. \qquad 00\ 01\ 10_{bin} \wedge\ 01\ 01\ 01_{bin} = 01\ 00\ 11_{bin}.$$

Inductive Step:*Assume $\mathbf{xor_1}(B^l_{d=3}) = A^l_{d=2} \cap \mathbf{xor_1}(B^l_{d=1}) = A^l_{d=0}$ holds for l. Since $\mathbf{xor_1}$ is an associative function, it is simple to prove that the initial statement holds for $l+1$, where $l \geq 1$.*

$$B^l_{d=3} \xrightarrow{\text{Expand B}} A^{l+1}_{d=2} \triangleright_{d=2} B^{l+1}_{d=3} \triangleright_{d=3} B^{l+1}_{d=3} \triangleright_{d=0} A^{l+1}_{d=0}$$

$$\Updownarrow$$

$$\begin{aligned}\mathbf{xor1}(B^l_{d=3}) &= \mathbf{xor1}(A^{l+1}_{d=2} \triangleright_{d=2} B^{l+1}_{d=3} \triangleright_{d=3} B^{l+1}_{d=3} \triangleright_{d=0} A^{l+1}_{d=0}) \\ &= \mathbf{xor1}(A^{l+1}_{d=2}) \triangleright_{d=3} \mathbf{xor1}(B^{l+1}_{d=3}) \triangleright_{d=2} \mathbf{xor1}(B^{l+1}_{d=3}) \triangleright_{d=1} \mathbf{xor1}(A^{l+1}_{d=0}) \\ &= B^{l+1}_{d=3} \triangleright_{d=3} A^{l+1}_{d=2} \triangleright_{d=2} A^{l+1}_{d=2} \triangleright_{d=1} B^{l+1}_{d=1}.\end{aligned}$$

In order to complete this proof, one would just need to replicate the induction step above for $\mathbf{xor_1}(B^l_{d=1}) = A^l_{d=0}$. Thus we consider this step as concluded, since $B^{l+1}_{d=3} \triangleright_{d=3} A^{l+1}_{d=2} \triangleright_{d=2} A^{l+1}_{d=2} \triangleright_{d=1} B^{l+1}_{d=1}$ is produced by $A^l_{d=2}$. □

Lemma 3.1 states that $\forall_{l\geq 1}\mathbf{xor_1}(B^l_{d=3}) = A^l_{d=2} \cap \mathbf{xor_1}(B^l_{d=1}) = A^l_{d=0}$. In fact, we are only interested in the first halve of this lemma, which states that the curve obtained by expanding $B_{d=3}$ l times is the same as the one obtained by expanding $A_{d=2}$ the same amount of times, after applying $\mathbf{xor_1}$ over it. Since $B_{d=3}$ and $A_{d=2}$ are present in the first expansion of this grammar, and each of these non-terminal symbols generate sub-curve that corresponds to the first and second quadrant of this curve, we conclude that the curve that represents the second quadrant is equal to the one contained by the first quadrant after applying $\mathbf{xor_1}$ over each of its elements.

Lemma 3.2. $\forall_{l\geq 1}\ \mathbf{xor_2}(B^l_{d=3}) = B^l_{d=1} \cap \mathbf{xor_2}(A^l_{d=2}) = A^l_{d=0}$.

Proof. This lemma is proven in a similar fashion to Lemma 3.1. Expanding the non-terminal symbols present in this lemma once, results in the same strings as in Formula 3.1 and 3.2. This strings are once again simplified, by Lemma 2.9, resulting on the strings present from Formula 3.3 to 3.6.

Base Case: *The curve generated by* $\mathbf{xor_2}(B^{L=1}_{d=3})$ *is equal to the one produced by* $B^{L=1}_{d=1}$*. The same is happens between* $\mathbf{xor_2}(A^{L=1}_{d=2})$ *and* $A^{L=1}_{d=0}$*. Implying the base case is verified:*

$$B_{d=1} \rightarrow\ 0\ 1\ 2_{dec} = 00\ 01\ 10_{bin}, \qquad A_{d=0} \rightarrow\ 1\ 0\ 3_{dec} = 01\ 00\ 11_{bin},$$

$$B_{d=3} \rightarrow\ 2\ 3\ 0_{dec} = 10\ 11\ 00_{bin}, \qquad A_{d=2} \rightarrow\ 3\ 2\ 1_{dec} = 11\ 10\ 01_{bin},$$

$$\mathbf{xor_2}(B_{d=3}) = B_{d=1} \qquad \mathbf{xor_2}(A_{d=2}) = A_{d=0}$$

$$\Longleftrightarrow \qquad \Longleftrightarrow$$

$$10\ 11\ 00_{bin} \wedge\ 10\ 10\ 10_{bin} = 00\ 01\ 10_{bin}. \qquad 11\ 10\ 01_{bin} \wedge\ 10\ 10\ 10_{bin} = 01\ 00\ 11_{bin}.$$

Inductive Step: *Analogously to Lemma 3.1, assume* $\mathbf{xor_2}(B^l_{d=3}) = B^l_{d=1} \cap \mathbf{xor_2}(A^l_{d=2}) = A^l_{d=0}$ *holds for* $l \geq 1$*. Keeping in mind that* $\mathbf{xor_2}$ *is also an associative function, one obtains the following equivalences:*

$$B^l_{d=3} \xrightarrow{\text{Expand } B} A^{l+1}_{d=2} \triangleright_{d=2} B^{l+1}_{d=3} \triangleright_{d=3} B^{l+1}_{d=3} \triangleright_{d=0} A^{l+1}_{d=0}$$

$$\Updownarrow$$

$$\begin{aligned}
\mathbf{xor2}(B^l_{d=3}) &= \mathbf{xor2}(A^{l+1}_{d=2} \triangleright_{d=2} B^{l+1}_{d=3} \triangleright_{d=3} B^{l+1}_{d=3} \triangleright_{d=0} A^{l+1}_{d=0}) \\
&= \mathbf{xor2}(A^{l+1}_{d=2}) \triangleright_{d=0} \mathbf{xor2}(B^{l+1}_{d=3}) \triangleright_{d=1} \mathbf{xor2}(B^{l+1}_{d=3}) \triangleright_{d=2} \mathbf{xor2}(A^{l+1}_{d=0}) \\
&= A^{l+1}_{d=0} \triangleright_{d=0} B^{l+1}_{d=1} \triangleright_{d=1} B^{l+1}_{d=1} \triangleright_{d=2} A^{l+1}_{d=2}.
\end{aligned}$$

□

Lemma 3.2 states that $\mathbf{xor_2}(B^l_{d=3}) = B^l_{d=1}$ and $\mathbf{xor_2}(A^l_{d=2}) = A^l_{d=0}$, for $l \geq 1$. In this case $B_{d=3}$ and $B_{d=1}$ represent the first and fourth quadrant, respectively, of a given HSFC where B is used as axiom. In its turn, $A_{d=2}$ and $A_{d=0}$ represent the first and the fourth quadrant of a given HSFC with axiom equal to A. Concluding that the curve contained by the fourth quadrant is equal to the one obtained by applying a $\mathbf{xor_2}$ curve located at the first quadrant. Note that similarly to what happens at the induction step of Lemma 3.1 we opted not to present the second halve of the proof for simplicity sake.

Lemma 3.3. *The values held by variables s_1, s_2, and s_3 depend on the parity of the curve iteration.*

Proof. Due to the incremental nature of the approaches, as described in Section 2.4 and 3.1, these algorithms produce a curve equivalent to one generated by a recursive Lindenmayer-System using axiom $B_{d=3}$, or $A_{d=2}$, if the starting direction is $d = 3$ for even iterations, or odd iterations if the starting direction is $d = 2$, respectively. Variable s_k represents the k^{th} step-forward command, $\triangleright$, present in the first expansion of these axioms:

$$A^l_{d=2} \xrightarrow{\text{Expand } A} B^{l+1}_{d=3} \overbrace{\triangleright_{d=3}}^{s_1} A^{l+1}_{d=2} \overbrace{\triangleright_{d=2}}^{s_2} A^{l+1}_{d=2} \overbrace{\triangleright_{d=1}}^{s_3} B^{l+1}_{d=1}, \tag{3.7}$$

$$B^l_{d=3} \xrightarrow{\text{Expand } B} A^{l+1}_{d=2} \underbrace{\triangleright_{d=2}}_{s_1} B^{l+1}_{d=3} \underbrace{\triangleright_{d=3}}_{s_2} B^{l+1}_{d=0} \underbrace{\triangleright_{d=0}}_{s_3} A^{l+1}_{d=3}. \tag{3.8}$$

Each of these commands is represented by a terminal symbol, meaning its value will remain unchanged for Lindenmayer-Systems starting with the same axiom. Thus, the values held by these values only depend on the starting production rule. □

Computing the values held by s_1, s_2, and s_3, at each iteration is quite straight forward. Lemma 3.3 presents the correct values each variable must hold for any given curve iteration. One could simply use an if-else statement and attribute these values to the respective variable. However, our will was to reduce the amount of pipeline breakage to the minimum.

Corollary 3.4. $s^{\mathcal{E}}_x \wedge 1 = s^{\mathcal{O}}_x \cap s^{\mathcal{O}}_x \wedge 1 = s^{\mathcal{E}}_x, \{x \in \{1, 2, 3\} \mid \mathcal{E} = \text{set of even numbers} \mid \mathcal{O} = \text{set of odd numbers}\}$.

Proof. Lemma 3.3 states that the correct values for each variable s_x are constant within even and odd curve iterations. A bijective function can be defined, mapping each element $s^{\mathcal{E}}_x$ to $s^{\mathcal{O}}_x$, and vice-versa. As shown in the following formulas:

$$s^{\mathcal{E}}_1 \wedge 1 = 3 \wedge 1 = 2 = s^{\mathcal{O}}_1, \tag{3.9}$$

$$s^{\mathcal{E}}_2, \wedge 1 = 2 \wedge 1 = 3 = s^{\mathcal{O}}_2, \tag{3.10}$$

$$s^{\mathcal{E}}_3, \wedge 1 = 1 \wedge 1 = 0 = s^{\mathcal{O}}_2. \tag{3.11}$$

Since every possible domain value was explored, from Formula 3.11 to 3.13, it is ensured that $s^{\mathcal{E}}_x \wedge 1 = s^{\mathcal{O}}_x$ will hold for every possible value of x. It is also ensured that $s^{\mathcal{O}}_x \wedge 1 = s^{\mathcal{E}}_x$ will hold, due to bit-wise exclusive-or's being an idempotent function. □

Corollary 3.4 made possible a more elegant and efficient solution. Instead of depending on if-else statements, the values for variable s_x are computed through bitwise exclusive-or operations, found in

Lines 7, 8, 9 of Algorithm 3.1. One just needs to initialize the values of variables s_x according to the parity of the first curve iteration and the starting production rule used by this Lindenmayer-System.

3.3 Time Analysis

As previously stated the time complexity of this algorithm is linear, $O(N)$. N is the number of entries present in a $n \times n$ matrix, where n is a power of 2. At each iteration of the for-loop three bit-wise exclusive-or functions are called, each with time complexity of $O(p)$. Thus resulting in a total time of $O(p)$ per iteration, all other operations run in constant time. In order to obtain a curve of level l the loop has to iterate a total of $l-1$ times. The i^{th} iteration of this loop computes a curve with $l = i+1$.

Lemma 3.5. *The time complexity of Algorithm 3.1 is $O(N)$.*

Proof. Let p_i be the i^{th} iteration of a given HSFC, and $a, b \in \mathbb{N}$ the number of constant time operations within the loop contained by $\mathbf{xor_x}$ functions and for-loop, line 5 of Algorithm 3.1, respectively. The total number of operations to compute a curve of level l is equal to the following sum:

$$S_l = (a\ ||p_1|| + b) + (a\ ||p_2|| + b) + ... + (a\ ||p_{l-2}|| + b) + (a\ ||p_l - 1|| + b).$$

Lemma 2.6 states that a curve of level l contains $4^l - 1$ directions. Thus $||p_i|| = 4^i - 1$:

$$S_l = (a(4-1)+b) + (a(4^2-1)+b) + ... + (a(4^{l-2}-1)+b) + (a(4^{l-1}-1)+b),$$
$$\Updownarrow$$
$$S_l = b(l-2) + \sum_{i=1}^{l-1} a(4^{i-1}-1).$$

Find a sum S'_l such that it is an upper-bound of S_l:

$$S'_l = b(l-2) + \sum_{i=1}^{l} a(4^{i-1}) \geq S_l = \sum_{i=1}^{l-1} bi + \sum_{i=1}^{l-1} a(4^{i-1}-1).$$

Since $\sum_{i=1}^{l} a(4^{i-1})$ is a geometric progression it is possible to simplify it further:

$$S'_l = b(l-2) + \sum_{i=1}^{l} a(4^{i-1}) = b(l-2) + \left(\frac{a(1-4^l)}{1-4}\right).$$

The level of any given HSFC is equal to $l = \log_4(N)$, resulting in:

$$\begin{aligned} S'_l &= b(l-2) + \left(\frac{a(1-4^l)}{1-4}\right) \\ &= b(\log_4(N)-2) + \left(\frac{a(1-4^{\log_4(N)})}{1-4}\right) \\ &= b(\log_4(N)-2) + \left(\frac{a(1-N)}{-3}\right) \\ &= b(\log_4(N)-2) + \left(\frac{a-aN}{-3}\right) \\ &= b(\log_4(N)-2) + \left(\frac{a(N-1)}{3}\right). \end{aligned}$$

Since a and b are constants, and $N > \log_4(N)$ we conclude that $S'_l \in O(N)$. Thus implying that $S_l \in O(N)$, due to S_l being majored by S'_l. □

3.4 Memory Analysis

The values used to represent a given direction in our approach are the same as the one used by the methods described in Section 2.4. All directions are represented by a set of integers and are stored in an integer array p. Each direction in p belongs to $\{0, 1, 2, 3\}$. We will now prove that our approach requires $O(N)$ memory, and that ideally it requires $2N - 2$ bits in order to store it in p.

Lemma 3.6. *Algorithm 3.1 ideally requires $2N - 2$ bits.*

Proof. *Let p be an array of integers, with size equal to $||p||$. Each entry of this array contains a direction p_i, where $i \in \mathbb{N}_0$ and $0 \leq i \leq ||p|| - 1$. Since $p_i \in \{0, 1, 2, 3\}$ every direction p_i needs at least 2 bits to be encoded, implying each direction requires $O(1)$ memory to be stored.*

$$\begin{aligned} 0_{dec} &= 00_{bin}, \\ 1_{dec} &= 01_{bin}, \\ 2_{dec} &= 10_{bin}, \\ 3_{dec} &= 11_{bin}. \end{aligned}$$

Once again, Lemma 2.6 states that a level l HSFC contains exactly $4^l - 1$ directions. Thus a total memory of $2(4^l - 1)$ bits is required to represent a curve with level equal to l. Since $l = log_4(N)$ the previous formula results in $2(N - 1)$ bits. Concluding the proof that Algorithm 3.1 requires $2N - 2$ bits, and thus a memory complexity of $O(N)$. □

The amount of memory required to run Algorithm 3.1 might look prohibitive for larger curves. In the following section we will present a modification to this approach that reduces its memory requirements.

3.5 Memory Optimization

As stated in the previous section, the most efficient implementation for an array p that holds $||p||$ elements will require $2N-2$ bits of memory. However it is possible to compress all information contained by p into a smaller array, as shown by Corollary 3.7.

Corollary 3.7. The amount of memory required to run this algorithm can be reduced to $N-2$ bits.

Proof. Assume that all possible primitive curves are indexed by their first element. It is possible to establish an injective relationship between their indexes and the actual curve, since the first element of each curve is unique:

$$0 \rightarrow B_{d=1}^{l=1} = 0\ 1\ 2, \tag{3.12}$$

$$1 \rightarrow A_{d=0}^{l=1} = 1\ 0\ 3, \tag{3.13}$$

$$2 \rightarrow B_{d=3}^{l=1} = 2\ 3\ 0, \tag{3.14}$$

$$3 \rightarrow A_{d=2}^{l=1} = 3\ 2\ 1. \tag{3.15}$$

If one would encode the path using these indexes instead of the whole primitive curve, one would just need to 2 bits to represent a given primitive curve, obviously followed by the required linking directions, encoded by s_x. In fact Algorithm 3.1 allows this modification very easily. It is possible to derive from Lemma 3.1 and Lemma 3.2 that the k^{th} element of the first quadrant is equal to the k^{th} elements of the second and third quadrant, after applying an exclusive-or operation by 1. The same is true for the first and fourth quadrant, but in this case an exclusive-or by 2 is applied. Implying all observations about our approach still hold true. Since the l^{th} iteration of a curve contains 4^{l-1} primitive curves, and each primitive curve can be encoded by 2 bits, the amount of memory required to store a given iteration of a HSFC is equal to:

$$\begin{aligned}
\overbrace{2N-2}^{\text{total \# bits}} - \overbrace{4(4^{l-1})}^{\text{\# removed bits}} &= 2N-2-4^{l} \\
&= 2N-2-4^{\log_4(N)} \\
&= 2N-2-N \\
&= N-2.
\end{aligned}$$

The amount of memory necessary to run the whole algorithm would be $N-2+24+O(1)$ bits. These 24 bits, is the exact amount of memory necessary to store all possible primitive curves. $\square$

3.6 Generalizing Space-Filling Curves

Prusinkiewicz et al. [7] present us a FASS-curve generator. This generator can draw space-filling curves that fill any $n \times n$ square grid. However generating these curves is only possible if a curve with the specific characteristics exists. This curve must be able to fill a $n' \times n'$ square grid. n' must be a multiple of 2. The starting and ending point of this curve must be located on opposite sides of the same row or same column. And the most important part is that this curve must first be found and hard-coded into our algorithm. All of these conditions provide a bad generalization/effort trade-off.

A different route was taken, we decided to adapt Algorithm 3.1 to the Nano-Programs approach, Section 2.4.3. Algorithm 3.1 is quite similar to the iterative Lindenmayer-System approach, Section 2.4.2. Both of these methods generate the same curve results within their processing loop. The differences arise outside of the scope of these loops, due to the computation of action code a. This action code can be easily simulated in our approach, through the use of a bit-mask array, where the value of each cell is either 0 or 1, stating whether the variable holding the direction value should remain unaltered between iterations, or if it should be changed. Computing this value increases greatly the overhead of this approach in comparison to Algorithm 3.1. Another array must be held in memory. Ideally this array would contain $N-1$ entries and each entry would occupy exactly 1 bit.

Both approaches, Algorithm 3.1 and the Nano-Programs, were combined as a proof of concept that our approach can be also used to generate pseudo-HSFC. In order to create more efficient approach, several optimization's would have to take place. The most important optimization would be taking advantage of curve being stored in memory. The same curve could be used to deal with the severe-asymmetrical cases found within the Nano-Programs implementation. Which would allow this approach to compute a pure HSFC only once, and therefore saving computational time.

4 Implementation Details

Contents

4.1 Bit Array

As previously observed, in Section 3.4, each direction contained by an HSFC can be encoded using only 2 bits. There is no primitive type in C able to contain a single direction without wasting storage space. Creating a structure holding only 2 values is also not a viable solution, since it is well known that manipulating primitive types is more efficient than user-defined types. We opt to store the curve in a bit-array composed by $\mathbf{char}$ type variables. The size of this primitive type variable is guaranteed to be equal to 8 bits, independently of which machine architecture or compiler is in use. Thus ensuring that our approach is portable. This bit array is represented by variable p, line 1 of Algorithm 4.1 , and has the following structure:

$$p = \{\underbrace{\overbrace{b_0,\ b_1}^{d_0},\ \overbrace{b_2,\ b_3}^{d_1},\overbrace{b_4,\ b_5}^{d_2},\ \overbrace{b_6,\ b_7}^{d_3}}_{\mathbf{char}_0},\ldots,\ \overbrace{b_{2N-5},b_{2N-4}}^{d_{N-2}},\ \overbrace{b_{2N-3},b_{2N-2}}^{d_{N-1}}\}.$$

Variables d_k and b_k, represent the k^{th} direction and bit respectively. This structure will be implemented in all approaches of Chapter 3, changing only in size, depending on which approach is used. The following sections will detail every step necessary to understand each phase of Algorithm 4.1, which is as close as possible to our C coded implementation, while remaining compact.

4.1.1 Allocating Memory

Allocating bit array p is quite straight forward. One only needs to know the iteration number of the desired HSFC, and which axiom to use. These parameters are represented by $\mathbf{int}$ variables L and d, respectively. As previously proven, the first version of XOR-Hilbert requires $2(4^L - 1)$ bits in order to represent the L^{th} iteration of a HSFC in memory, implying

$$||p|| \geq \frac{2(4^L - 1)}{8}. \tag{4.1}$$

Since $||p||$ represents the number of $\mathbf{char}$ variables contained by our bit array. We conclude that the number of $\mathbf{char}$ variables needed to store iteration L of any HSFC is:

$$||p|| = 4^{L-1}. \tag{4.2}$$

The memory optimized version of XOR-Hilbert requires $4^L - 2$ bits in order to represent the same curve as above. The number of $\mathbf{char}$ variables needed to represent this curve is exactly

$$||p|| = \frac{4^L - 2}{8}. \tag{4.3}$$

Algorithm 4.1: XOR-Hilbert

```
Input: $L$, and $d$
Output: $p$
begin
1   $p$ := malloc(sizeof(char) * N_CHARS($L$));
    /* starting production rule is $A_d = 2$*/                */
2   if $d = 2$ then
3       $s_1 := 128$;
4       $s_2 := 192$;
5       $s_3 := 0$;
6       $p := 14$;
    /* starting production rule is $B_d = 3$*/                */
7   else if $d = 3$ then
8       $s_1 := 192$;
9       $s_2 := 128$;
10      $s_3 := 64$;
11      $p := 27$;
12  for $l := 0; l < L; l := l + 1$ do
13      for $c := 0; c < 4^{l-1} - 1; c := c + 1$ do
14          $p[c + 4^{l-1}] := p[c] \wedge$ XOR1111;
15          $p[c + 2(4^{l-1})] := p[c] \wedge$ XOR1111;
16          $p[c + 3(4^{l-1})] := p[c] \wedge$ XOR2222;
17      $p[4^{l-2} - 1 + 4^{l-2}] := p[4^{l-1} - 2] \wedge$ XOR0111;
18      $p[4^{l-2} - 1 + 2(4^{l-2})] := p[4^{l-2} - 1] \wedge$ XOR0111;
19      $p[4^{l-2} - 1 + 3(4^{l-2})] := p[4^{l-2} - 1] \wedge$ XOR0222;
20      $p[4^{l-2} - 1] := p[4^{l-2} - 1] || s_1 \wedge 64$;
21      $p[2(4^{l-2}) - 1] := p[2(4^{l-2}) - 1] || s_2 \wedge 64$;
22      $p[3(4^{l-2}) - 1] := p[3(4^{l-2}) - 1] || s_3 \wedge 64$ ;
```

However $||p||$ must be the smallest integer able to contain this curve, implying that

$$||p|| \geq \frac{4^{L-1}}{2}. \tag{4.4}$$

The amount of $\mathbf{char}$ variables needed to represent a curve using this approach is an halve of the amount required by the first version of XOR-Hilbert. The last approach, Nano-Programs embedded with XOR-Hilbert, requires the computation of a HSFC with iteration number equal to $L - 1^{th}$, as seen in Section 2.4.3. Concluding that the number of allocated $\mathbf{char}$ variables is equal to

$$||p|| \geq \frac{2(4^{L-1} - 1)}{8}. \tag{4.5}$$

Once again $||p||$ must be an integer, thus

$$||p|| = 4^{L-2}. \tag{4.6}$$

This study concludes that the amount of $\mathbf{char}$ variables our bit array must contain in order to be able to store the L^{th} iteration of a given HSFC is equal to 4^{L-1} $\mathbf{char}$ variables for the regular XOR-Hilbert version, 4^{L-2} $\mathbf{char}$ variables for the memory-optimized version, and 4^{L-2} $\mathbf{char}$ variables for the version that makes use of Nano-Programs.

4.1.2 Directions Layout

One of the most important implementation choice was deciding how should the directions, that compose a HSFC, lay within our bit array p. The implications that arise from this decision are propagated to every part of our algorithm. Two different layouts were explored. The first layout is based on the approach taken by the Nano-Programs, Section 2.4.3. This method uses shift-right operations in order to obtain a given direction from p, thus destroying our bit array. The second layout makes use of bit-masks and bit-wise and operations to retrieve a given direction. Implying the values held by p remain unchanged while processing the curve. It is important to note that it is also possible to preserve our bit-array while processing it with the first layout. However an extra variable must be used to store a subset of the path. The previously destructive operations are then applied to this temporary variable.

4.1.2.A Nano-Program Layout

As seen in Section 2.4.3, the directions are laid out in diminishing order, in relation to the order they should be processed. Meaning that the 2 least-significant bits of a given Nano-Program encode the first direction to be processed. While the 2 most-significant bits, of this Nano-Program, encode the last direction to be processed. Laying out p in the same fashion as a Nano-Program results in the following scheme:

$$p = \underbrace{\{d_3,\ d_2,\ d_1,\ d_0\}}_{\mathbf{char}_0},\ \underbrace{\{d_7,\ d_6,\ d_5,\ d_4\}}_{\mathbf{char}_1}, ... \tag{4.7}$$

Variable d_k, where $k \in [0, 4^L - 2]$, represents the order by which directions should be processed. In order to obtain direction d_k, one has to apply a shift-right operation of $2(k \bmod 4)$ bits to $p[\lfloor k/4 \rfloor]$.

4.1.2.B Non-Destructive Layout

As previously stated, this layout was designed to retrieve directions from p using bit-masks. Which implies that p will remain unchanged during the processing step. Another benefit of using bit-masks is that these variables allow directions to be sorted by order of traversal within a $\mathbf{char}$ variable, thus presenting a layout that is more logical, in terms of $\mathbf{char}$ traversal:

$$p = \underbrace{\{d_0,\ d_1,\ d_2,\ d_3\}}_{\mathbf{char}_0},\ \underbrace{\{d_4,\ d_5,\ d_6,\ d_7\}}_{\mathbf{char}_1}, ... \tag{4.8}$$

The bit-masks implemented in our program are the following:

$$\mathbf{XOR0} := 11000000_{\mathbf{bin}} = 192_{\mathbf{dec}}, \tag{4.9}$$

$$\mathbf{XOR1} := 00110000_{\mathbf{bin}} = 48_{\mathbf{dec}}, \tag{4.10}$$

$$\mathbf{XOR2} := 00001100_{\mathbf{bin}} = 12_{\mathbf{dec}}, \tag{4.11}$$

$$\mathbf{XOR3} := 00000011_{\mathbf{bin}} = 3_{\mathbf{dec}}. \tag{4.12}$$

In order to obtain the value of direction d_k, one must use bit-mask $\mathbf{XOR}k$, where $k \in [0, 3]$, to perform a bit-wise and operation between this mask and $p[\lfloor k/4 \rfloor]$. Finally a bit-wise shift-right operation by $2(k \bmod 4)$ is applied, to the previously computed value, thus obtaining d_k.

Both of these layouts were compared and no significant performance difference was found. However, if one would run a program that require the same curve more than twice, re-utilizing this curve could be seen as significant run-time optimization. Henceforth, all implementation details will assume the first layout is in use, except when said otherwise. This decision will simplify future explanations, and ensures greater consistency between different approaches.

4.1.3 Initialization

As depicted by Algorithm 4.1, before running this algorithm the user can decide which axiom should be used. The possibility of deciding the starting production rule of a given HSFC was required in order to embed XOR-Hilbert within FUR-Hilbert 2.6. This was not necessary for the remaining approaches. However we decided to implement this feature in all of the other methods, since it should not affect our running time. The axiom choice is encoded by variable d. If $d = 2$, then the starting production rule will be $A_{d=2}$, and variables s_1, s_2, and s_3 will assume the same values as Formula 3.9. Otherwise, the starting production rule will be $B_{d=3}$, and variables s_1, s_2, and s_3 will assume the values of Formula 3.8.

4.2 Building The Next Curve Iteration

In Section 3.1 was stated that it is possible to compute all quadrants, and linking directions of the next iteration of a given HSFC, based on the current iteration of this curve, using bit-wise exclusive-or and concatenation operations. In our implementation all of these functions and operations are contained within lines 13 to 22 of Algorithm 4.1. If we view this set of statements as a function, then it would receives the following variables as input:

- Variable l, the number of the current iteration,
- variables s_1, s_2, and s_3, the linking directions,
- and lastly a reference to p, our bit array.

Identifying which entries of p belong to a given quadrant of the next iteration is well known. Since every $\mathbf{char}$ variable contains 4 directions, it is possible to rewrite the previous layout, Formula 4.7, in such a way that each cell of p is represented by a $\mathbf{char}$ variable:

$$\begin{aligned} p = \{ & \underbrace{\mathbf{char}_0, \cdots, \mathbf{char}_{4^{l-2}-1}}_{\mathbf{quadrant_1}}, \\ & \underbrace{\mathbf{char}_{4^{l-2}}, \cdots, \mathbf{char}_{2(4^{l-2})-1}}_{\mathbf{quadrant_2}}, \\ & \underbrace{\mathbf{char}_{2(4^{l-2})}, \cdots, \mathbf{char}_{3(4^{l-2})-1}}_{\mathbf{quadrant_3}}, \\ & \underbrace{\mathbf{char}_{3(4^{l-2})}, \cdots, \mathbf{char}_{4^{l-1}-1}}_{\mathbf{quadrant_4}} \}. \end{aligned}$$

Variables s_1, s_2, and s_3 are found in the first 2 bits of $\mathbf{char}_{4^{l-2}-1}$, $\mathbf{char}_{2(4^{l-2})-1}$, and $\mathbf{char}_{3(4^{l-2})-1}$, respectively. This layout is in conformance with the first method of processing, from Section 2.4.3, and the regular version of XOR-Hilbert. Corollary 3.4, implies that the layout scheme for the memory optimized version of this algorithm should be the following:

$$
\begin{aligned}
p = \{ & \underbrace{\mathbf{char}_0, \cdots, \mathbf{char}_{\frac{4^{l-2}}{2}-1},}_{\mathbf{quadrant_1}} \\
& \underbrace{\mathbf{char}_{\frac{4^{l-2}}{2}}, \cdots, \mathbf{char}_{2\frac{4^{l-2}}{2}-1},}_{\mathbf{quadrant_2}} \\
& \underbrace{\mathbf{char}_{2\frac{4^{l-2}}{2}}, \cdots, \mathbf{char}_{3\frac{4^{l-2}}{2}-1},}_{\mathbf{quadrant_3}} \\
& \underbrace{\mathbf{char}_{3\frac{4^{l-2}}{2}}, \cdots, \mathbf{char}_{\frac{4^{l-1}}{2}-1}\}}_{\mathbf{quadrant_4}}
\end{aligned}
$$

Basically the only change between the $\mathbf{xor}$ function of these two approaches is how do we compute the boundaries of each quadrant. This computation is computed based on the current iteration number, or level, l. In this case, variables $s1$, $s2$, and $s3$ will be found in the first 2 bits of $\mathbf{char}_{\frac{4^{l-2}}{2}-1}$, $\mathbf{char}_{4^{l-2}-1}$, and $\mathbf{char}_{3\frac{4^{l-2}}{2}-1}$, respectively.

From this point onward, all details will be explained based on the regular version of XOR-Hilbert. Since the only adaption needed to turn one algorithm into another are the computations concerning the boundaries of each quadrant.

4.2.1 Computing and Storing New Quadrants

Now that the boundaries that of each quadrant are known, it is possible to apply the proper bit-mask, and concatenate the newly computed quadrant entries into p. Four different bit-masks are needed in order to compute the new quadrants:

$$
\begin{aligned}
\mathbf{XOR0111} &:= 00010101_{bin} = 21_{dec}, \\
\mathbf{XOR1111} &:= 01010101_{bin} = 85_{dec}, \\
\mathbf{XOR0222} &:= 00101010_{bin} = 42_{dec}, \\
\mathbf{XOR2222} &:= 10101010_{bin} = 170_{dec}.
\end{aligned}
$$

Masks $\mathbf{XOR1111}$ and $\mathbf{XOR2222}$, lines 14 to 16 of Algorithm 4.1, are used to perform bitwise exclusive-or operation to the first $4^{l-2}-1$ $\mathbf{char}$ variables of p. The values obtained through this operation is then attributed to the correct entry of p. The computation of these entries is based on the offset between the first quadrant and the other three quadrants:

$$\forall_{k\in[0,4^{l-2}-1]}\ p[4^{l-2}+k] := p[k] \wedge \mathbf{XOR1111}, \tag{4.13}$$

$$\forall_{k\in[0,4^{l-2}-1]}\ p[2(4^{l-2})+k] := p[k] \wedge \mathbf{XOR1111}, \tag{4.14}$$

$$\forall_{k\in[0,4^{l-2}-1]}\ p[3(4^{l-2})+k] := p[k] \wedge \mathbf{XOR2222}. \tag{4.15}$$

Masks $\mathbf{XOR0111}$ and $\mathbf{XOR0222}$, lines 17 to 19 of Algorithm 4.1, are used to compute the value of the last $\mathbf{char}$ variable, in each of the last three quadrants:

$$p[2(4^{l-2})-1] := p[4^{l-2}] \wedge \mathbf{XOR0111}, \tag{4.16}$$

$$p[3(4^{l-2})-1] := p[4^{l-2}] \wedge \mathbf{XOR0111}, \tag{4.17}$$

$$p[4^{l-1}-1] := p[4^{l-2}] \wedge \mathbf{XOR0222}. \tag{4.18}$$

This corner case exists due to variables s_1, s_2, and s_3 being added to the curve through bit-wise or operations, represented by operator ||. It is needed to ensure that the bits that will hold this variables remain equal to 0.

4.2.2 Computing and Storing Linking Directions

The operations, used to add this variables to our curve, lines 20 to 22 of Algorithm 4.1, are represented by:

$$p[2(4^{l-2}-1)] := p[2(4^{l-2}-1)]\ ||\ s_1; \tag{4.19}$$

$$p[3(4^{l-2}-1)] := p[3(4^{l-2}-1)]\ ||\ s_2; \tag{4.20}$$

$$p[4^{l-1}-1] := p[4^{l-1}-1]\ ||\ s_3; \tag{4.21}$$

Reminding the layout we are using to process this curve. The values held by these linking direction variables have to be padded by 6 zeroes, on the right-side, in order to only affect the 2 most-significant bits of the `char` variable that will contain them. Thus the value held by each variable s_x must be equal to:

$$s_1^{\mathcal{O}} := 11000000_{bin} = 192_{dec}, \qquad s_1^{\mathcal{E}} := 10000000_{bin} = 128_{dec}, \tag{4.22}$$

$$s_2^{\mathcal{O}} := 10000000_{bin} = 128_{dec}, \qquad s_2^{\mathcal{E}} := 11000000_{bin} = 192_{dec}, \tag{4.23}$$

$$s_3^{\mathcal{O}} := 01000000_{bin} = 64_{dec}. \qquad s_3^{\mathcal{E}} := 00000000_{bin} = 0_{dec}. \tag{4.24}$$

After adding these variables to p, it is ensured that currently a HSFC with level equal to l is stored in our bit array. In order to process the next iteration of this HSFC, a bit-wise exclusive-or between each s_x and literal integer 64. Our program is ready to repeat this whole process, computing the next iteration of this curve, if deemed necessary.

4.3 Processing the Curve

Once that the final iteration of this curve is produced and stored in our bit array, one must compute the next entry of our grid. This new entry is computed based on the current pair of coordinates (i, j) and direction, d_k, where $0 \leq k \leq 4^L - 1$, onto which we must step. The direction values must be fetch and processed in crescent order in respect to k. The fetching and processing methods were already detailed in Section 2.4, which results in Algorithm 4.2.

4.3.1 Loop-Unrolling

For simplicity sake, it was decided to implement the `process_curve` function using an unrolled-loop, as shown in Algorithm 4.2. This allowed us to save some time during the development process of our approach, since it allowed to access the correct entries of bit-array p with ease. We are aware that using the Loop-Unrolling technique will boost the run-time performance of our approach while comparing XOR-Hilbert with other approaches. In order to obtain a fair comparison, it was decided to measure the performance of all approaches enabling the compiler to automatically perform an unroll of every loop. The compiler options will be further detailed in Section 4.6.

4.3.2 User-Defined Functions

Our implementation only computes and processes a given HSFC, thus defining which path of traversal should be applied to a given grid. The adaptation of this traversal to a given application must be done by the user. Nonetheless, if a naive solution is already implemented, adapting it to our implementation should be fairly simple. In order to adapt the path of traversal created by XOR-Hilbert one must replace all `//process_entry` comments by the desired operation or sequence of operations. Algorithm 4.3 depicts how one can transform Algorithm 4.2 into the cumulative sum of every entry within matrix A and store it in variable S.

Algorithm 4.2: Process Curve

```
Input: $P$, and $C$
begin
    $(i, j) := (0, 0)$;
    //process_entry;
    for $\underline{c = 0; c < C - 1; c := c + 1}$ do
        $p := P[c]$;
        $d := p \bmod 4$;
        $p := p >> 2$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        //process_entry;
        $d := p \bmod 4$;
        $p := p >> 2$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        //process_entry;
        $d := p \bmod 4$;
        $p := p >> 2$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        //process_entry;
        $d := p \bmod 4$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        //process_entry;
    $p := P[C - 1]$;
    $d := p \bmod 4$;
    $p := p >> 2$;
    $i := (d - 2) \bmod 2$;
    $j := (d - 1) \bmod 2$;
    //process_entry;
    $d := p \bmod 4$;
    $p := p >> 2$;
    $i := (d - 2) \bmod 2$;
    $j := (d - 1) \bmod 2$;
    //process_entry;
    $d := p \bmod 4$;
    $i := (d - 2) \bmod 2$;
    $j := (d - 1) \bmod 2$;
    //process_entry;
```

Algorithm 4.3: Cumulative Sum

```
Input: $P$, and $C$
begin
    $(i, j) := (0, 0)$;
    $S := S + A[i][j]$;
    for $\underline{c = 0; c < C - 1; c := c + 1}$ do
        $p := P[c]$;
        $d := p \bmod 4$;
        $p := p >> 2$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        $S := S + A[i][j]$;
        $d := p \bmod 4$;
        $p := p >> 2$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        $S := S + A[i][j]$;
        $d := p \bmod 4$;
        $p := p >> 2$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        $S := S + A[i][j]$;
        $d := p \bmod 4$;
        $i := (d - 2) \bmod 2$;
        $j := (d - 1) \bmod 2$;
        $S := S + A[i][j]$;
    $p := P[C - 1]$;
    $d := p \bmod 4$;
    $p := p >> 2$;
    $i := (d - 2) \bmod 2$;
    $j := (d - 1) \bmod 2$;
    $S := S + A[i][j]$;
    $d := p \bmod 4$;
    $p := p >> 2$;
    $i := (d - 2) \bmod 2$;
    $j := (d - 1) \bmod 2$;
    $S := S + A[i][j]$;
    $d := p \bmod 4$;
    $i := (d - 2) \bmod 2$;
    $j := (d - 1) \bmod 2$;
    $S := S + A[i][j]$;
```

4.4 Generalizing the Curve

As explained in Section 3.6, in order to generalize a HSFC we opted to embed the XOR-Hilbert approach into FUR-Hilbert. To achieve this generalization we decided to emulate the behaviour of the iterative Lindenmayer-System using our algorithm. This adaptation is depicted by Algorithm 4.4. The values held by variable d, within these two approaches, only differs outside of the scope of their respective for-loops. This value is used to obtain the index of a given Nano-Program in Algorithms 2.6 and 4.4. If it is possible to compute this value, then it will also be possible to generalize our approach with Nano-Programs. The iterative approach of a Lindenmayer-System requires variables a, h, and l, which represent the action-code, the iterations steps and the level of a given direction within a HSCF. Computing the action code a, recurring only to the variables present in our approach seemed to be challenging at first, since our algorithm does not contain a variable h, present in Algorithms 12 and 2.6. However, obtaining the values of these variables that are associated with each direction was quite straightforward and similar to the computation of the path contained by our bit array. Each action code, a, is closely tied to what we previously designated by linking directions, referenced by variables s_k. In [1], the value of a given action code is ranged by $[1, 3]$. We opted to change the notation for greater consistency with our approach. Each action code will now be ranged between $[0, 2]$. A given variable a holds value k, iff the direction this variable is suppose to transform was generated by linking direction variable s_k. Since all directions contained by a given curve were once a linking direction of a previous iteration of this curve we reach the following schema for the second iteration of a HSFC:

$$p = \{\underbrace{s_2,\ s_1,\ s_0}_{l=1}, \overbrace{s_0}^{l=2}, \underbrace{s_2,\ s_1,\ s_0}_{l=1}, \overbrace{s_1}^{l=2}, \underbrace{s_2,\ s_1,\ s_0}_{l=1}, \overbrace{s_2}^{l=2}, \underbrace{s_2,\ s_1,\ s_0}_{l=1}\}$$

Variable l is known within the scope of the for-loop, found at line 12 of Algorithm 2.6. Now that we know the values of variables a and l, it is finally possible to compute $(\texttt{isOdd}(l-1)\ \wedge\ a = 1)$. This computation only needs to be performed once for each pair (s_k, l). Instead of computing this operation each time a given entry is being processed, it was decided to compute it only once, and propagate this value through bit-array X, which will held the value of this operation for each direction within our curve. Now that these values are stored in X, we can retrieve and use them in lines 16 and 21 of Algorithm 4.4, respectively. Thus completely emulating the behaviour of the iterative Lindenmayer-System, within and outside of its main for-loop. Since this approach makes use of a new bit-array the total amount of memory needed is now doubled. Requiring a total amount of $2(4^{L-2})$ $\mathbf{char}$ variables to store both bit-arrays in memory.

Algorithm 4.4: XOR-Hilbert Embedded into Nano-Programs

```
Input: $(I, J)$, $(i_{max}, j_{max})$
begin
    $d := 0$;
    $t := \lfloor \log_2 n \rfloor$;
    $D :=$ XOR-Hilbert(n,d);
    $A :=$ ComputeActions(n,d);
    while $h < 2^{2t-2}$ do
        $r := \lfloor \frac{(I+1)n}{t} \rfloor - \frac{I \cdot n}{t}$;
        $D := \lfloor \frac{(J+1)m}{t} \rfloor - \frac{J \cdot m}{t}$;
        $P :=$nanoPrograms$[r][s][d]$;
        while $P \neq 0$ do
            processObjectPair$(i, j)$;
            $c := P \bmod 4$;
            $P := \lfloor P/4 \rfloor$;
            $j := j + ((c - 1)) \bmod 2$;
            $i := i - ((2 - c)) \bmod 2$;
        $d :=$getNextDirection$(D)$;
        $a :=$getNextActionCode$(A)$;
        $j := j + ((d - 1) \bmod 2)$;
        $i := i + ((d - 2) \bmod 2)$;
        $J := J + ((d - 1) \bmod 2)$;
        $I := I + ((d - 2) \bmod 2)$;
        $d := d \wedge a$;
```

4.5 Approach Restrictions

Depending on the system architecture, every program whose memory complexity is larger than constant, will eventually incur in a run-time error due to lack of memory. The memory complexity of our approaches grows linearly in order to N.However N grows exponentially, implying our system will exhaust the available memory rather quickly. This limitation is bounded to hardware rather than implementation. All of these approaches make use of $\textbf{unsigned int}$ variable n. This variable stores the side length of our grid. If variable n is greater than $2^{64} - 1$ will result in improper program behaviour. If $n = 2^{63}$ it is implied our approaches will compute a curve that fills a square grid with 4^{63}, thus incurring in a curve containing $2(4^{63} - 1)$ bits. Concluding that most modern systems will exhaust available memory before getting close to this number.

4.6 Approach Comparison

All approaches were implemented using C programming language. The code implementation of FUR-Hilbert is provided by a link found in [1]. Within the inner-most for-loop of this approach one can find the implementation of the iterative Lindenmayer-System. The following experiments have been performed on Intel Xeon E7- 4830 CPU with 2.13GHz and 8 cores. The cache hierarchy of this CPU consists on levels L1i (32KB), L1d (32KB), L2 (256Kb), and L3 (24576KB). The operative system in use is Debian GNU/Linux 9 (stretch). The compiler in use is gcc, version 6.3.0. All programs were compiled with flag -O2, due to the explicit unrolled loop present in XOR-Hilbert. The optimizations performed by this compiler can be found in its documentation [18]. The main reason for using -O2 instead of -O3 was that -O3 optimizations actually slowed down the performance of every approach, in comparison to -O2. All approaches were run 5 times and averaged. Figures 4.1 and 4.2, as Tables 4.1 and 4.2, make use of the following labels: XOR-Hilbert (XOR), memory optimized XOR-Hilbert (MEM), XOR-Hilbert generalization (NAN), FUR-Hilbert (FUR), and finally the iterative Lindenmayer-System (LDN). These abbreviations will be carried for the rest of this document. All implementations can be found at `https://github.com/JoaoAlves95/XOR-Hilbert/`. Keep in mind this code might be object of re-factorization.

4.6.1 Run-Time Analysis

As it can be seen in the log-plot present at Figure 4.1, all of our approaches incur in an initial overhead in comparison to FUR-Hilbert and the iterative Lindenmayer-System approach. The Lindenmayer-System based approach is faster than our approaches for iterations smaller than 6, i.e. matrices with dimensions smaller than 64×64. On the other hand, the run-time performance of FUR-Hilbert is surpassed by our approach for curves with with level higher than 8. From iteration number 10 onward, the XOR-Hilbert approach is at least 7.10 times faster than the iterative Lindenmayer-System approach and 1.27 times faster than the FUR-Hilbert approach. The peak performance achieved by XOR-Hilbert is at most 7.42 times faster than the Lindenmayer-System approach, and 1.33 times faster than FUR-Hilbert. All alternative versions of XOR-Hilbert present better performance than FUR-Hilbert and the Lindenmayer-System, as depicted by Table 4.1.

4.6.2 Memory Analysis

The amount of memory used by each approach was measured by using command `/usr/bin/time -f %M`, which returns the maximum resident set in memory. As expected all of our approaches present a linear memory growth, depicted by the log-plot in Figure 4.2, in contrast to the approaches presented by [1], which require a constant amount of memory. This growth difference starts to be noticeable after iteration number 9, for XOR-Hilbert and its memory optimized version. The growth rate of our take onto Nano-Programs is only noticeable after iteration number 10 of an HSFC. As expected both of the alternative versions to XOR-Hilbert require less memory. The memory required to run XOR-Hilbert for iteration

15 is 27.67 times larger than the memory required to run the previous approach. Table 4.2 details the memory comparison between all approaches.

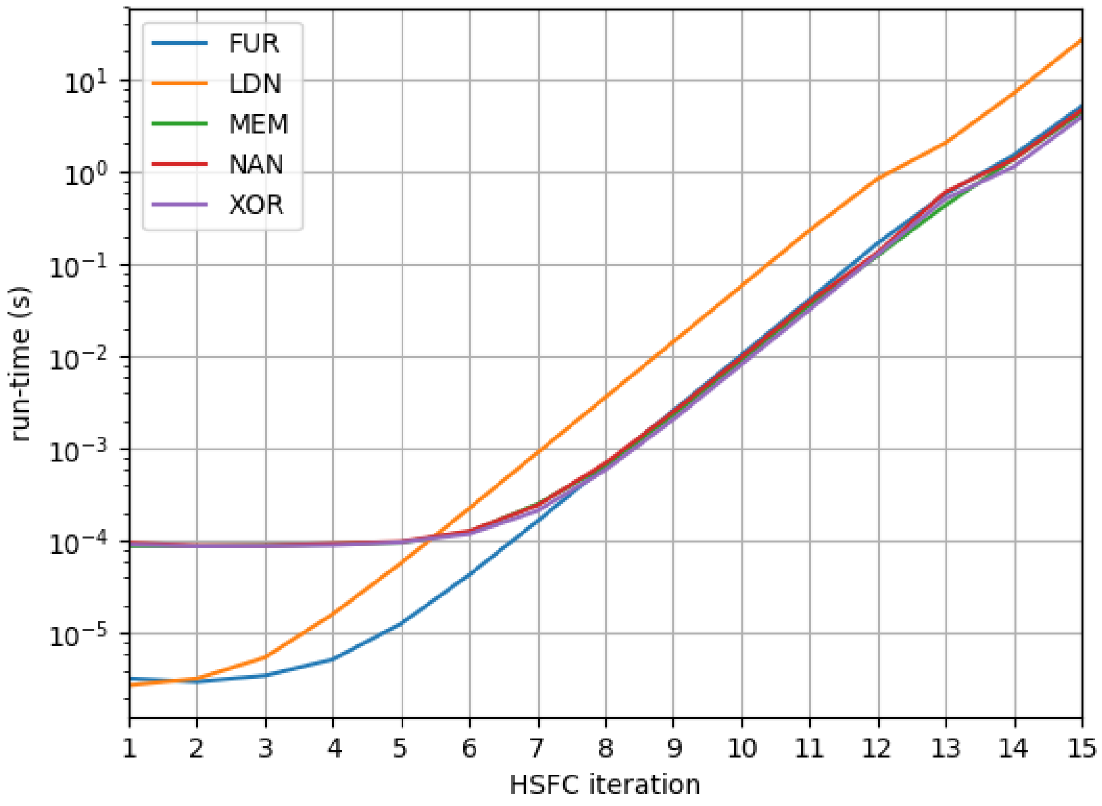


Figure 4.1: Log-plot depicting linear run-time growth for every approach.

Table 4.1: Run-time comparison between different approaches.

Iteration	XOR	MEM	NAN	FUR	LDN
1	9.1750e-05	8.9e-05	9.55e-05	3.25e-06	2.75e-06
2	8.825e-05	8.8499e-05	8.9500e-05	3e-06	3.25e-06
3	8.8749e-05	9e-05	8.9500e-05	3.5e-06	5.5e-06
4	9.025e-05	9.2500e-05	9.375e-05	5.25e-06	1.6250e-05
5	9.725e-05	9.65e-05	9.8500e-05	1.2750e-05	5.7749e-05
6	0.000119	0.000124	0.000128	4.3e-05	0.000225
7	0.000212	0.00025	0.000243	0.000163	0.000898
8	0.000585	0.000646	0.000695	0.000649	0.003583
9	0.002064	0.002306	0.002505	0.002574	0.014346
10	0.008115	0.008975	0.009721	0.010312	0.057507
11	0.031826	0.035584	0.038778	0.041166	0.230775
12	0.126627	0.122598	0.132845	0.167088	0.833475
13	0.507307	0.431860	0.600483	0.573634	2.041121
14	1.125117	1.366718	1.357902	1.503230	7.001975
15	3.894231	4.347082	4.63702	5.0928702	26.61183

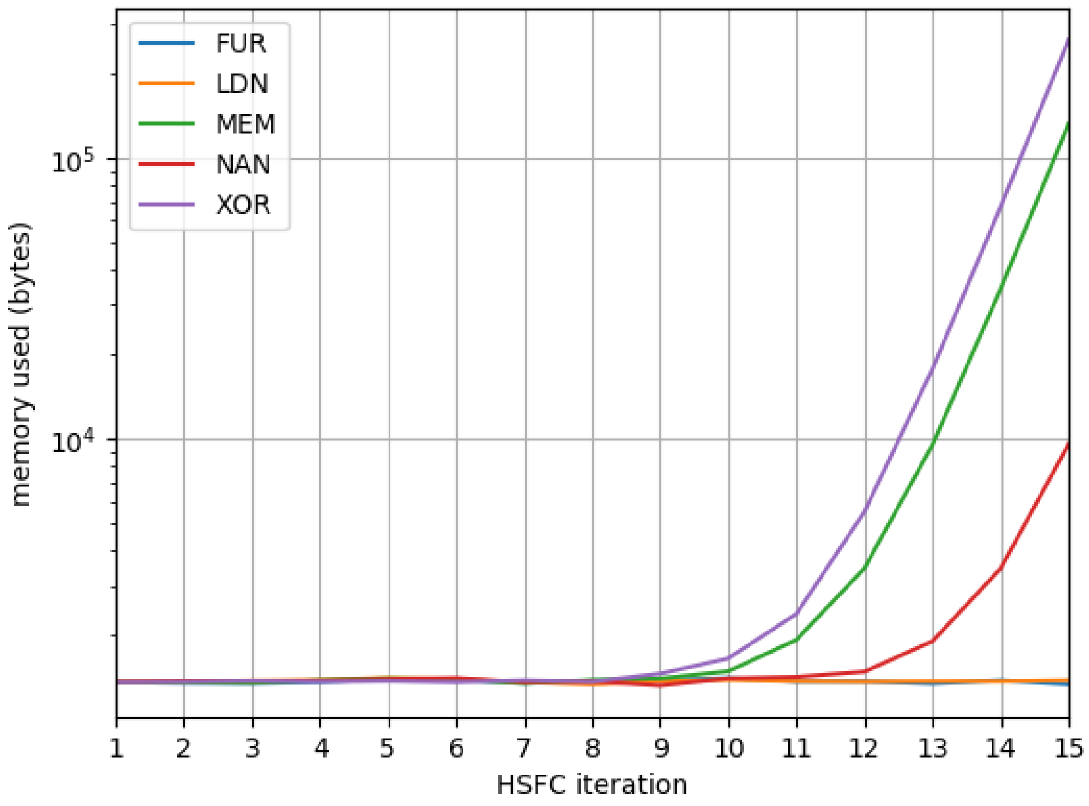


Figure 4.2: Log-plot depitcting the memory growth of each approach.

Table 4.2: Memory usage comparison between different approaches.

Iteration	XOR	MEM	NAN	FUR	LDN
1	1365	1362	1367	1364	1374
2	1368	1361	1379	1355	1371
3	1380	1357	1378	1352	1382
4	1370	1386	1381	1367	1389
5	1378	1406	1398	1397	1412
6	1365	1397	1411	1389	1398
7	1386	1352	1364	1356	1365
8	1369	1389	1374	1382	1341
9	1462	1405	1329	1384	1375
10	1657	1491	1406	1409	1390
11	2384	1928	1423	1373	1375
12	5507	3465	1486	1374	1370
13	17748	9533	1907	1350	1375
14	66957	34123	3456	1386	1375
15	263547	132499	9588	1342	1386

5 Applications and Evaluation

Contents

Algorithm 5.1: Naive Transposition

```
Input: $n$
begin
1   for $i := 0; i < n; i := i + 1$ do
2       for $j := 0; j < n; j := j + 1$ do
3           $B[j][i] := A[i][j]$;
```

5.1 Experimental Environment

In [1], the authors do not only show how to truly compute an Hilbert Space-Filling Curve in linear time, but also how to apply the path of traversal generated by this curve to make naive approaches into cache-oblivious algorithms. The focus of this chapter is to describe each of the applications that were embedded in our approaches followed by their respective time and cache analysis. All following operations were ran in a single core of the previously described machine, Section 4.6.

The time measurements of each application were computed using `clock` function, present in `C time.h` library. Only the execution of the application was measured, the setup time for matrices or other variables unrelated to our application were neglected. All plots depicting the run-time comparison between different approaches were calculated by applying $100(\frac{\texttt{run-time(Naive)}}{\texttt{run-time(App)}} - 1)$. Each bar of this plot represents the performance gain of HSFC approaches in comparison to the naive one. It is important to note that all approaches based on HSFC present an instruction-wise overhead. The performance of these approaches will only surpass the naive approach once the number of cache-misses avoided compensates the amount of extra instructions required to run this approaches. Thus justifying why the run-time of the naive approach might be smaller for the first iterations of our analysis.

Cache measurements were computed using Cachegrind [19]. This tool emulates the behaviour of a 2-level hierarchy cache. The two levels of this cache were setup in order to have the same characteristics of the L1 and LLC caches present in our machine. Unfortunately it is not possible to add an intermediate cache-level.

5.2 Out-of-Place Matrix Transposition

The first application to be tested was an out-of-place matrix transposition. This application scans a given matrix A while storing its entries in output matrix B. At the end of this application B shall contain a matrix equivalent to the transpose of A. The most naive implementation of an out of place matrix transposition, for a $n \times n$ matrix, should not look a lot different than the nested for-loop found in Algorithm 5.1.

Note that by following this method it is implied that one of the matrices will be traversed in row-major

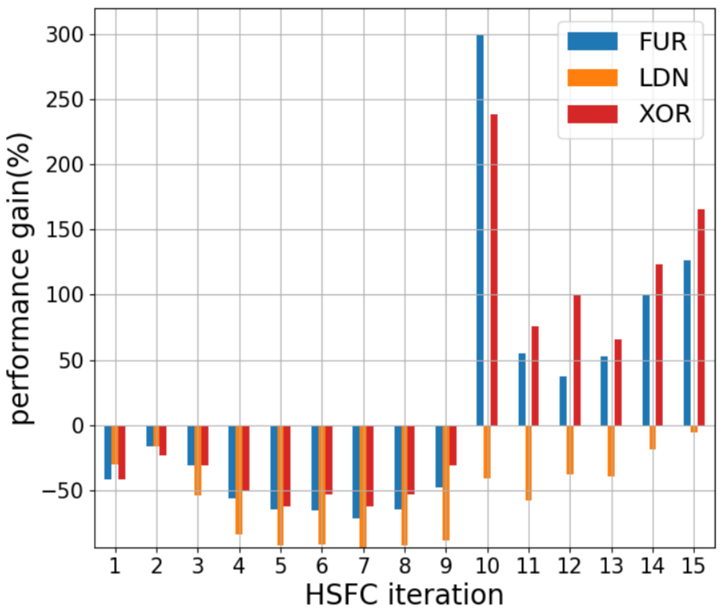


Figure 5.1: Matrix transposition performance gain.

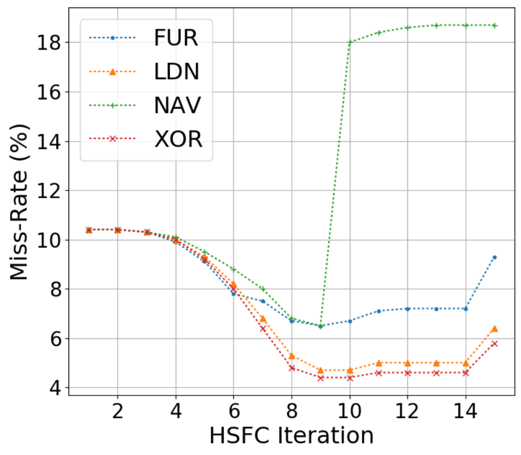


Figure 5.2: Miss-rate for matrix transposition reading operations in cache L1d.

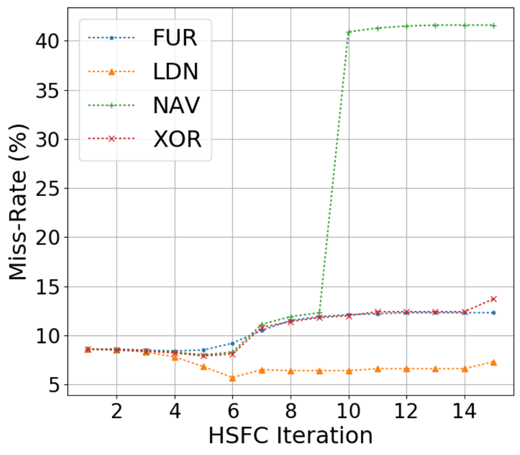


Figure 5.3: Miss-rate for matrix transposition writing operations in cache L1d.

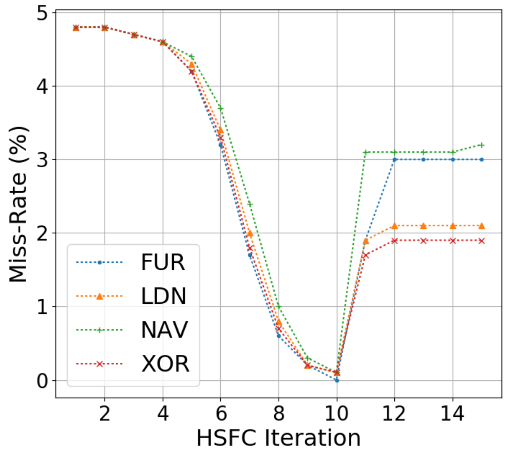


Figure 5.4: Miss-rate for matrix transposition reading operations in cache LLd.

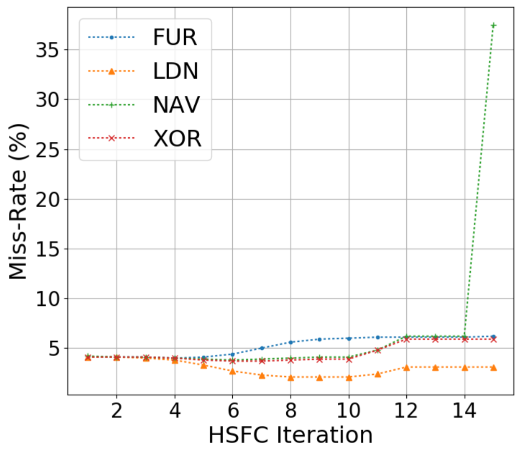


Figure 5.5: Miss-rate for matrix transposition writing operations in cache LLd.

scan, while the other is scanned using column-major order. In other words, one of these matrices will be traversed in a sub-optimal manner. However, traversing A following a path of traversal produced by an Hilbert curve, implies that the path used to traverse B will also be an Hilbert curve. More specifically if A is traversed using an HSFC generated by axiom $A_{d=2}$, then the path by which B is traversed is an HSFC that uses $B_{d=3}$ as the initial production rule. If both of these matrices are laid in either row-major or column-major layout, then the cache hit-rate should be optimal for both matrices.

5.2.1 Run-Time Analysis

As previously stated the benefits of these loops will only be observable when the number of cache-misses avoided compensates the instruction overhead of each approach. This tipping point can be observed at iteration number 10, where XOR-Hilbert presents a gain of 239% in performance, while FUR-Hilbert presents a performance gain of 300%, in comparison to the naive approach (NAV). In the context of this study. It is our belief that difference in performance reflects a corner case phenom due to the characteristics of the machine in use. From iteration number 2 onward our approach shows greater performance than any other approach, including FUR-Hilbert except for iteration number 10. The overhead presented by the Lindenmayer-System approach would require a larger test domain to overcome the running time of the naive approach.

5.2.2 Cache Analysis

Using the ideal cache-model, present in Section 2.1.1, it would be expected that the approaches based on an HSFC would present a stable cache-miss ratio for larger iterations of this application. It would also be expected that the cache-miss ratio of the naive approach would grow linearly, from a given point in time. None of these assumptions was observed, Figures 5.2 to 5.5. One of the reasons behind these results is the association level of the cache in use. Since we are not using a fully-associative cache, each memory block will have a predefined set of cache lines where it can be allocated. Implying a linear growth of collision while hashing each memory block to the respective cache-line. This strays the LRU policy even further form the optimal replacement policy, used in the ideal cache-model, since at each hash collision our cache is potentially evicting a memory block that will still be used. However, the cache-miss ratios, present in Figures 5.2, 5.3, and 5.5, do not show any significant decay in cache performance between our approach and FUR-Hilbert. Actually, our approach presents a performance increase in relation to FUR-Hilbert, except at cache level L1d-Write, where FUR-Hilbert presents a smaller cache-miss ratio for iteration numbers 7 and 15. Figure 5.4 depicts the miss-rate for last level of cache, for write operations. The cache behaviour of this approach for LL-Write is slightly better than the one presented by our approach. However, from iteration number 11 onward, our approach outperforms the

Algorithm 5.2: Transposed Floyd-Warshall

```
Input: $n$
begin
1   for $i := 0; i < n; i := i + 1$ do
2       for $j := 0; j < n; j := j + 1$ do
3           for $k := 0; k < n; k := k + 1$ do
4               if $A[j][i] + A[i][k] < A[j][k]$ then
                    $A[j][k] = A[j][i] + A[i][k];$
```

cache behaviour of FUR-Hilbert. XOR-Hilbert presents a constant miss-rate of 1.9%, while FUR-Hilbert presents a constant miss-rate of 3%. The Lindenmayer-System based approach shows an improved cache behaviour for write operations, Figures 5.4 and 5.5. This approach does not make use of any memory to represent the curve, suggesting that the matrix segments used by this algorithm can remain in cache longer, and thus presents a smaller miss-rate than the other HSFC based approaches.

5.3 Floyd-Warshall Algorithm

Another algorithm used in [1] to evaluate the performance of this cache-oblivious loop was the Floyd-Warshall algorithm [20]. This algorithm solves the *all pair shortest path* problem with time complexity of $O(n^3)$ and memory complexity $O(n^2)$, since this algorithm operates over an adjacency matrix with dimension $n \times n$. The pseudo-code depicting the main loop of the naive approach can be observed by the triply-nested loop present in Algorithm 5.2. As its possible to observe in the inner-most loop of this algorithm, we are assuming that the adjacency matrix representing the graph was previously transposed. In fact no transposition operation was implemented, in this section we are only evaluating the performance of the Floyd-Warshall algorithm.The HSFC approaches also implement the optimization based on transposition.

5.3.1 Run-Time Analysis

As expected the first iterations of HSFC approaches present little to no performance gain. The adjacency matrix fits completely in cache implying no misses were avoided. From iteration 5 to 9 it is possible to observe that the performance gain of the HSFC approaches grows almost linearly. XOR-Hilbert presents the best performance gain within this interval, peaking at iteration number 8 with a performance gain of 65%. Similarly to what happens in Section 5.2, it is our belief that iteration number 9 is the result of a phenom identical to the one observed in iteration 10 of the matrix transposition operation. However the approach that benefited from this corner case was XOR-Hilbert. The only moment FUR-

Hilbert outperformed our approach was in iteration 10, with a performance gain equal to 71% while our approach presents a performance gain of 67%. At iteration number 12, the matrix does not fit in cache anymore. All HSFC approaches present similar results, being slightly outperformed by XOR-Hilbert with a performance gain of 93%.

5.3.2 Cache Analysis

Opposed to what would be expected, the miss-ratio of the naive approach is smaller than the miss-ratio of HSFC based approaches for writing operations. This lead us to the conclusion that the main reason for the poor performance of the naive approach is the amount of cache-misses it incurs during read operations. In fact after iteration number 6, Figure 5.7, the amount of cache-misses presented by the naive approach grows drastically from 0.5% to 4.4%. Something similar happens after iteration number 12 for the last level of cache, Figure 5.9. The adjacency matrix does not fit in cache anymore, thus increasing the cache-miss ratio incurred by reading operations to 2.2%. The remaining approaches do not show any change on their miss-rates, presenting a constant ratio of 0%.

All HSFC based approaches present fairly similar rates of cache-misses. The largest difference happens in iteration number 6, Figure 5.8, where XOR-Hilbert presents a miss-rate of 2.4% for writing operations within L1d cache, while FUR-Hilbert presents a miss-rate of 1.1%. This difference is then minimized for larger iterations.

5.4 Matrix Multiplication

In [1], one of the applications used to test the efficiency of FUR-Hilbert was the matrix multiplication [20]. In order to perform this analysis we took a different approach. Instead of just performing a simple matrix multiplication we opted to measure the performance of a matrix transposition followed by the respective multiplication.The time complexity of this operation is $O(n^3)$, since matrix transposition runs in $O(n^2)$ and matrix multiplication runs in $O(n^3)$. The memory complexity is $O(n^2)$, since this application requires four $n \times n$ matrices to be stored in memory.

This measurements were compared against two different naive approaches. The first approach follows the triply nested-loop depicted by Algorithm 5.3. Implying no optimizations were implemented. This approach will be used as the base case for our comparison. The second naive approach consists in the transposition of matrix B followed by the multiplication procedure, detailed by Algorithm 5.4. This optimization allows matrix B to be scanned in an optimized fashion, since subsequent visited entries will be found in adjacent memory positions [21], in contrast with the first naive approach. From Figures 5.11 to 5.15 this optimized approach will be referenced by label TRN. The HSFC based approaches compute the transposition of matrix B using the applications presented in Section 5.2.

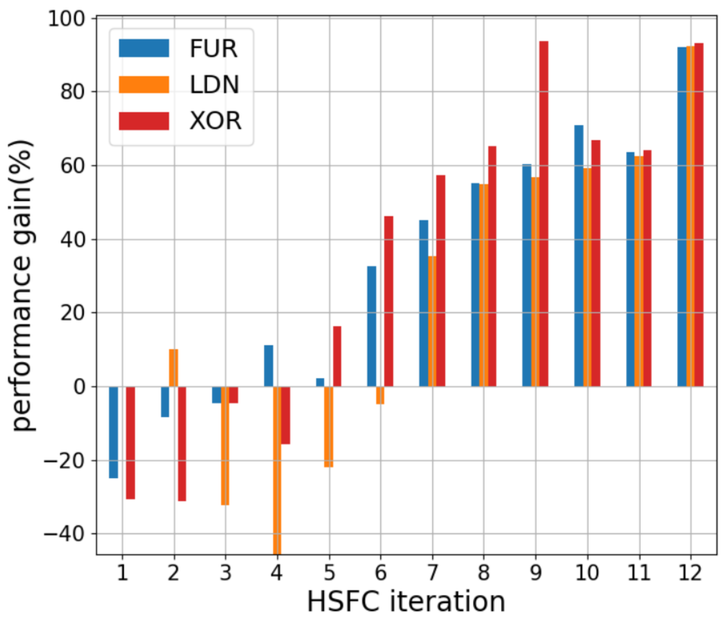


Figure 5.6: Floyd-Warshall algorithm performance gain.

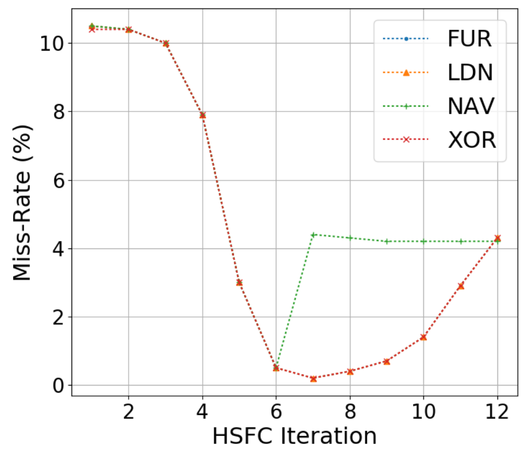


Figure 5.7: Miss-rate for Floyd-Warshall algorithm reading operations in cache L1d.

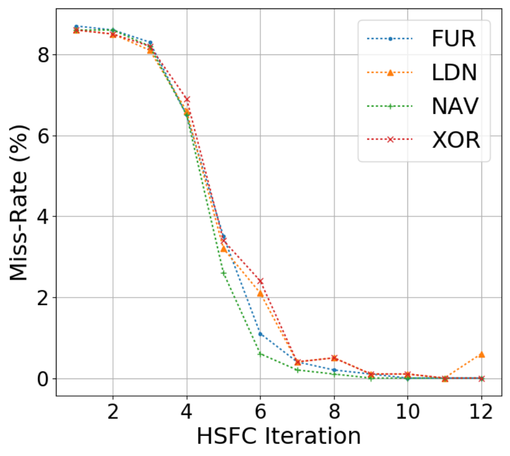


Figure 5.8: Miss-rate for Floyd-Warshall algorithm writing operations in cache L1d.

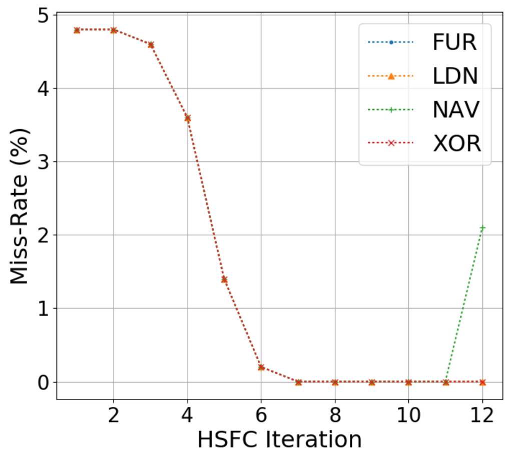


Figure 5.9: Miss-rate for Floyd-Warshall algorithm reading operations in cache LLd.

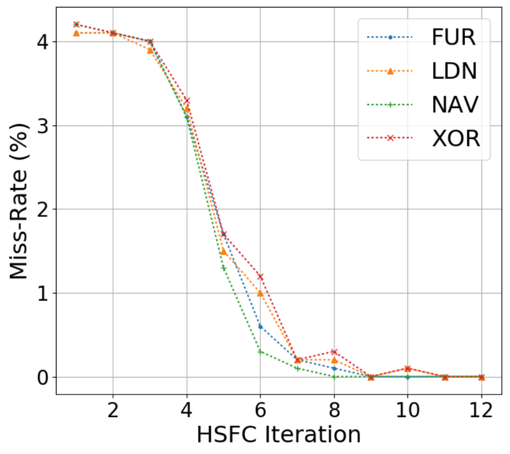


Figure 5.10: Miss-rate for Floyd-Warshall algorithm writing operations in cache LLd.

Algorithm 5.3: Naive Multiplication

```
Input: $n$
begin
1   for $i := 0; i < n; i := i + 1$ do
2       for $j := 0; j < n; j := j + 1$ do
3           for $k := 0; k < n; k := k + 1$ do
4               $C[i][j] := A[i][k] * B[k][j];$
```

Algorithm 5.4: Transposed Multiplication

```
Input: $n$
begin
1   for $i := 0; i < n; i := i + 1$ do
2       for $j := 0; j < n; j := j + 1$ do
3           for $k := 0; k < n; k := k + 1$ do
4               $C[i][j] := A[i][k] * B[j][k];$
```

5.4.1 Run-Time Analysis

Similarly to what happens with Floyd-Warshall algorithm, the first iterations of the HSFC approaches presents little or no gain in performance, Figure 5.11. Due to the small size of the matrices, it is possible to completely store them in cache. From iteration 6 to 11 we can observe a slightly increase in performance for all approaches. Again, this behaviour was expected. Our best guess is that the matrices can still fit completely in cache, however the higher levels of cache should be full, thus most of the matrices lines are found in the lower levels of cache. Since the naive approach is required to scan a memory block per access to matrix B a large amount of memory blocks must be swapped between higher and lower levels of cache, degrading the performance of the naive approach. At iteration number 12 we can finally observe the benefits of preserving locality using an Hilbert curve. The performance of HSFC based approaches keeps growing, while the performance of the naive approach optimized by a transposition decreases. Our approach, XOR-Hilbert, only presents better performance than FUR-Hilbert at iteration number 9 and 10.

5.4.2 Cache Analysis

This application behaves identically to Flowyd-Warshall, Section 5.3. The rate of cache-misses caused by writing operations is similar for every approach, Figures 5.13 and 5.15. Once again leading us to the conclusion that the main cause for the poor performance of the naive approaches is the amount cache-misses caused by reading operations. It is possible to observe a particularly interesting case in Figure 5.12. The naive approach, Algorithm 5.3, has a lower but still close number of operations in comparison to the naive approach optimized by a transposition, Algorithm 5.4. At iteration number 9 the performance of these two approaches is fairly similar. However, at iteration number 10 the cache-miss rate of the naive approach increases from 5% to almost 50%. At this point in time the miss-rate of the naive approach in cache LL is still equal to 0%. Implying no data is being fetched from memory, but still from the last level of cache. Fetching data from different cache-levels is still much faster than fetching data from memory. Even though the miss-rate of the naive approach is only large for the L1d cache, this was enough to make this approach almost 7 times slower than its transposed counterpart.

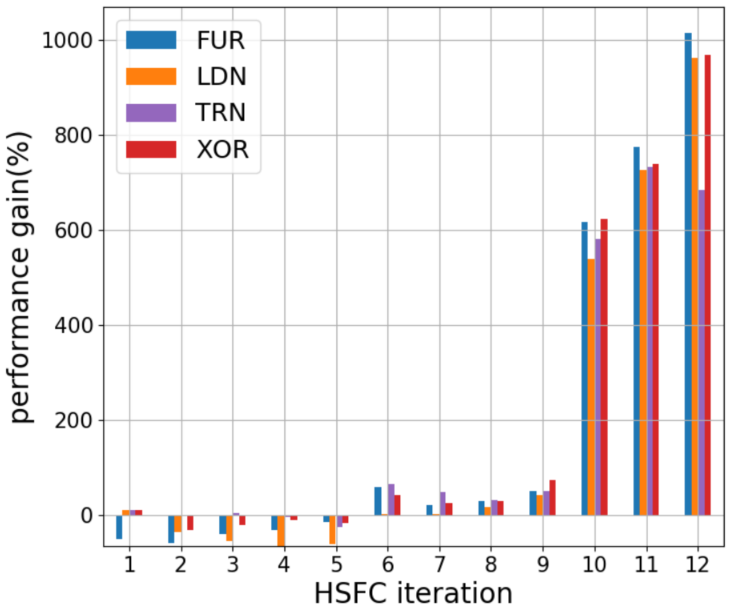


Figure 5.11: Matrix multiplication performance gain.

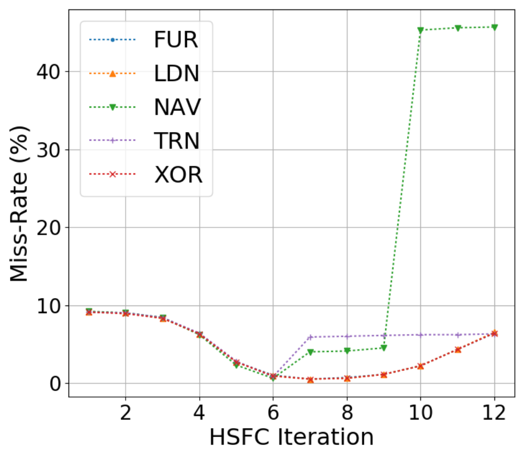


Figure 5.12: Miss-rate for matrix multiplication algorithm reading operations in cache L1d.

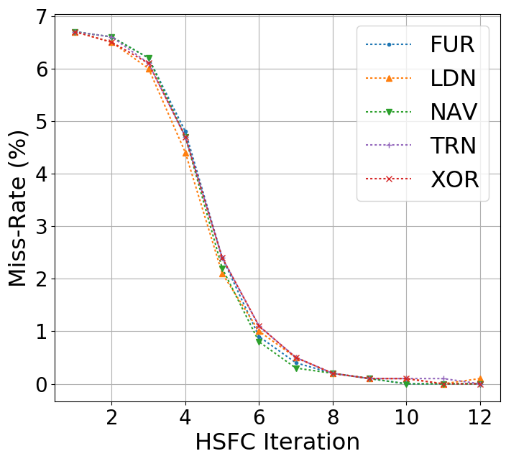


Figure 5.13: Miss-rate for matrix multiplication algorithm writing operations in cache L1d.

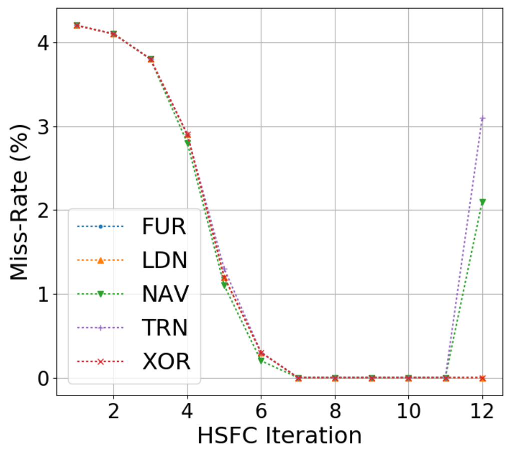


Figure 5.14: Miss-rate for matrix multiplication algorithm reading operations in cache LLd.

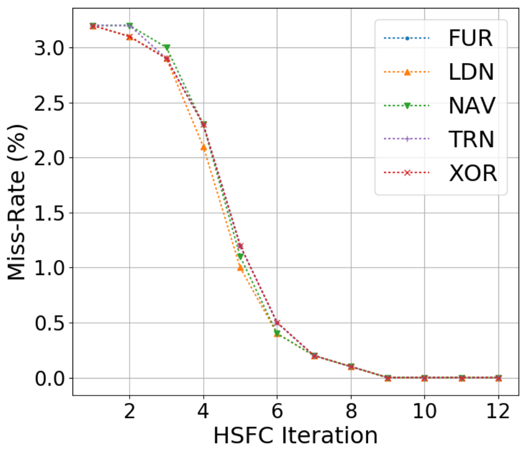


Figure 5.15: Miss-rate for matrix multiplication algorithm writing operations in cache LLd.

6

Conclusion

Contents

6.1 Conclusions

This work describes several efficient alternatives to compute a loop based on Hilbert Space-Filling Curves. Our main approach is competitive against currently existing solutions. Even though it might seem paradoxical, the biggest strength of XOR-Hilbert is the fact that the whole curve is stored in memory, after being computed. Therefore allowing for run time amortization of subsequent function calls, that make use of the same or smaller iterations of theses curves. It is also important to understand that these approaches were designed for problems that most of the time will require a full traversal of one or more matrices. Therefore the memory requirements of these algorithms will be $O(m.n)$, which minimize the limitations of our approach.

These loops seem ideal to run algorithms that inherently use some sort of matrix traversal and present no data dependencies, such as matrix multiplication. It is also important to note, as stated by *Böhm et al.* [1], that these loops provide a more generalized interface than most Basic Linear Algebra Subprograms (BLAS) implementations. These implementations are bounded to the problem they seek to solve, while these loops can be used in different contexts with minimal modifications. We also performed a comparison between these approaches and BLAS in order to compare the performance of Cache-Oblivious and Cache-Conscious algorithms. However this study was of no avail, BLAS is not only a Cache-Conscious algorithm, it also employs vectorization techniques. Implying no possible comparison between the approaches discussed in this paper and its BLAS equivalent. Even though these loops improve the spatial and temporal locality of the CPU cache, additional techniques must be implemented in order to obtain competitive results against state of the art BLAS. We believe it is possible to obtain similar performance gains to the matrix transposition if one employs some sort of blocking technique.

6.2 Future Work

Now that this thesis is concluded we can identify the most relevant continuations for this work. It seems interesting to conduct a different study in order to establish what other algorithms can be improved by employing these HSFC based loops. Even more interesting would be designing alternative versions for algorithms known to have data dependencies, like LCS and other dynamic programming algorithms, in a manner that would allow these new versions to be ran through one of these loops.

We profiled our approach, using perf, and around 20% of the total run-time of XOR-Hilbert is used to compute to compute the curve. The curve computation of XOR-Hilbert, in contrast to currently existent approaches, seems like an ideal candidate for hand-coded vectorization. Our approach computes the curve using only two types of exclusive-or operations, with no data dependencies within the same iteration of our main for-loop. The logical next step to this study would be to implement a vectorized version of XOR-Hilbert that also applies blocking techniques, and then compare the performance of this approach to the one of FUR-Hilbert and to a state-of-the-art BLAS, such as MKL-BLAS [22].

In this paper we already demonstrated that it is possible to generalize the curves produced by our approach making use of Nano-Programs, Sections 3.6 and 4.4. As previously stated, this approach was only implemented as a proof of concept. In a future paper we will present a version of XOR-Hilbert designed for Manycore architectures. Since this architecture has several CPU caches available there is no need to traverse a matrix using a simple curve. Instead we can split the original matrix in different sub-matrices and reduce the traversal problem to a tiling problem. Each of these tiles will be traversed by a single HSFC, thus improving the locality of CPU cache associated with each tile. The details concerning matrix splicing are not yet clear. As previously stated we can reuse a previously computed curve to traverse a matrix with a number of entries equal or smaller than the amount of directions contained by this curve. If we limit the dimensions of the generated curve with an upper-bound, we will probably be able to trade memory for run-time performance.

Bibliography

[1] C. Böhm, M. Perdacher, and C. Plant, “Cache-oblivious loops based on a novel space-filling curve,” in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 17–26.

[2] G. Arroz, J. Monteiro, and A. Oliveira, “Arquitectura de computadores–dos sistemas digitais aos microprocessadores,” 2009.

[3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, “Cache-oblivious algorithms,” in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[4] J. W. Suh and Y. Kim, *Accelerating MATLAB with GPU computing: A primer with examples*. Newnes, 2013.

[5] J. Iliffe, “The use of the genie system in numerical calculation,” in *International Tracts in Computer Science and Technology and Their Application*. Elsevier, 1961, vol. 2, pp. 1–28.

[6] E. D. Demaine, “Cache-oblivious algorithms and data structures,” *Lecture Notes from the EEF Summer School on Massive Data Sets*, vol. 8, no. 4, pp. 1–249, 2002.

[7] P. Prusinkiewiczl, A. Lindenmayert, and D. Fracchia, “Synthesis of space-filling curves on the square grid,” 1991.

[8] H. V. Jagadish, “Linear clustering of objects with multiple attributes,” in *ACM SIGMOD Record*, vol. 19, no. 2. ACM, 1990, pp. 332–342.

[9] D. Voorhies, “Space-filling curves and pace-filling curves and a measure of coherence,” *Graphics Gems II*, p. 26, 1991.

[10] A. Cavalcanti and D. Dams, *FM 2009: Formal Methods: Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009, Proceedings*. Springer, 2009, vol. 5850.

[11] Hilbert curve - rosetta code. (visited on 2019-05-21). [Online]. Available: https://rosettacode.org/wiki/Hilbert_curve#C

[12] P.Prusinkiewicz, “L-systems: from the theory to visual models of plants,” 1996.

[13] T. Norvell, “A short introduction to regular expressions and context free grammars,” *Project report, Nov*, vol. 5, 2002.

[14] H. Abelson and A. Sessa, *The computer as a medium for exploring mathematics*. Mit Press, 1980.

[15] S. E. Anderson, “Bit twiddling hacks, 2005,” *URL: https://graphics. stanford. edu/seander/bithacks. html# InterleaveBMN (visited on 13/08/2019)*.

[16] R. Goldman, S. Schaefer, and T. Ju, “Turtle geometry in computer graphics and computer-aided design,” *Computer-Aided Design*, vol. 36, no. 14, pp. 1471–1482, 2004.

[17] C. S. ISO, “Iec 9899,” *Programming languages-C*, 1999.

[18] GNU Project. (2019) Using the gnu compiler collection (gcc). (visited on 2019-08-15). [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc

[19] Cachegrind: a cache and branch-prediction profiler. (visited on 2019-08-30). [Online]. Available: http://valgrind.org/docs/manual/cg-manual.html

[20] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[21] U. Drepper, “What every programmer should know about memory,” *Red Hat, Inc*, vol. 11, p. 2007, 2007.

[22] Intel, “Blas and sparse blas routines,” *Intel® Math Kernel Library for C*, Sep 2019, (visited on 2019-10-5). [Online]. Available: https://software.intel.com/en-us/mkl-developer-reference-c-blas-and-sparse-blas-routines