Cache-Oblivious Nested Loops Based on Hilbert Curves

João Nuno Estevão Fidalgo Ferreira Alves
joao.ferreira.alves@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2019

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 non-stateless, thus allowing the re-utilization of previously computed Hilbert Space-Filling Curves. Which in allows the time required to compute another iteration of this curve, or other operations, to be amortized.

Keywords: cache-oblivious algorithms, Hilbert curve, space-filling curves, cache-oblivious for-loop

1. Introduction

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 a 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 novelle space-filling curve (SFC) able to traverse matrices with any given dimension. This work depicts 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.

2. Background

It is well-known that traversing a given matrix M using a row-major [11] or column-major scan [11] has a large impact on the amount of cache-misses. Thus degrading the performance of any program that misuses any of the previous scans. It is only intuitive that using a different traversal on a given matrix might impact the performance of an algorithm. Its our belief this was the motivation behind the work of *Böhm et al.*[1].

2.1. Hilbert Curve Properties and Restrictions

Böhm et al.[1] define a mapping function called the Hilbert Inverse $\mathcal{H}^{-1}(x)$, which maps each point of an HSFC curve to an entry of our grid. Being the HSFC part of the FASS family[9], it is guaranteed that by enumerating any two successive arguments, this function will return two grid entries that are horizontally or vertically adjacent to each other. Thus preserving locality between adjacent entries of this grid. 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 only 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 [12], i.e. locality, than Peano.

2.2. Computing the Hilbert Curve

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 [2], or other approaches that scan the bits of current iteration value h, as [10]. 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-Systems [8] adaptation. Our study will only be focused on the approaches presented in [1].

2.2.1 Recursive Lindenmayer-System

These systems are built upon an appropriate Context-Free Grammar [7], composed by terminal symbols, non-terminal symbols, and production rules. This approach receives as input the desired level of recursion or iteration of a given HSFC, and returns a string describing the directions of a 2-dimensional Hilbert curve. The Lindenmayer-System used to compute an arbitrarily large iteration of an HSFC is defined in [1].

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). 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 $d := (d+3) \bmod 4$. Terminal symbol π can be seen as a getter method for the current grid entry coordinates, (i, j). Finaly, terminal symbol $\triangleright$, which symbolizes a step-forward command on current direction $d \in \{0, 1, 2, 3\}$, can be encoded by the following statements:

$$\begin{aligned}
&\triangleright_{d=0}\ \mathit{command} \Rightarrow j := j+1, &&\text{//move right}\\
&\triangleright_{d=1}\ \mathit{command} \Rightarrow i := i+1, &&\text{//move up}\\
&\triangleright_{d=2}\ \mathit{command} \Rightarrow j := j-1, &&\text{//move left}\\
&\triangleright_{d=3}\ \mathit{command} \Rightarrow i := i-1. &&\text{//move down}
\end{aligned}$$

The two production-rules that define this system are:

$$A \rightarrow \pi | \ominus B \triangleright \oplus A \triangleright A \oplus \triangleright B \ominus, \qquad (1)$$
$$B \rightarrow \pi | \oplus A \triangleright \ominus B \triangleright B \ominus \triangleright A \oplus. \qquad (2)$$

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 is known a proper encoding is known for every production rules and terminal symbol it is quite simple to translate this Lindenmayer-System to code, as depicted by Algorithms 1. The encoding for production-rule B follows the same logic as A.

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].

2.2.2 A Novelle Iterative Lindenmayer-System and FUR-Hilbert

In order to overcome the drawbacks presented by the approaches in Section 2.2, *Böhm et al.*[1]

Algorithm 1: Rule A

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

developed two different approaches that compute a complete HSFC in linear-time. The first approach emulates the recursion stack of a traditional Lindenmayer-System based on the observation that the bit-pairs used to represent h in *4-dic* detail which non-terminal symbol was expanded within a given production rule.

The second approach called FUR-Hilbert can generate a pseudo HSFC able to fill any rectangular grid. This algorithm splices the original grid originating several sub-grids. The order by which these sub-grids are processed is defined by the iterative Lindenmayer-System. Finally a Nano-Program is applied to each of these sub-grids resulting in a FASS curve able to traverse any rectangular grid.

These Nano-Programs are small pre-processed FASS curves 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 [5]. The domain of possible directions values is equal to $\{0, 1, 2, 3\}$. Converting a given direction to a coordinate-pair of our grid follows the same formula as Section 2.2.1. 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.

3. 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 2. This algorithm produces the next iteration of a given HSFC in a incremental fashion. Thus, the first quadrant of the next iteration is always equal to 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 2. For now we are neglecting the `process_curve`. It is already established from Section 2.2 that it is possible to process the whole curve in $O(N)$.

3.1. Proof of Correction

In order to prove the correction of Algorithm 2 several lemmas were stated and demonstrated in the

Algorithm 2: XOR-Hilbert

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

main document of this thesis. In this extended abstract we will only state and explain what conclusions we can take from these lemmas.

Lemma 3.1.

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

Lemma 3.1 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.

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

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. 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.

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

Corollary 3.3.1.

$$s^{\mathcal{E}}_x \wedge 1 = s^{\mathcal{O}}_x \cap s^{\mathcal{O}}_x \wedge 1 = s^{\mathcal{E}}_x, \{x \in \{1,2,3\}\}.$$

Combining Lemma 3.3 and Corollary 3.3.1 allow us to determine the value of variable s_x in a more efficient and elegant fashion than incurring on `if-else` statements. $s^{\mathcal{O}}$ and $s^{\mathcal{E}}$ state whether we are computing the value for variable s_x in a odd or even iteration of our loop, line 5 of Algorithm 2. Corollary 3.3.1 states that a bit-wise exclusive or by 1 establishes a mapping between $s^{\mathcal{E}}_x$ and $s^{\mathcal{O}}_x$. Since the exclusive or operation is idempotent there is no need for auxiliary variables to compute a given s_x, as seen in lines 7 to 9 of Algorithm 2.

Lemma 3.4. *The time complexity of Algorithm 2 is $O(N)$.*

Lemma 3.4 was proven by majoring the sum of operations required to compute iteration number L of a given HSFC by a geometric progression known to have a linear growth rate.

Lemma 3.5. *Algorithm 2 ideally requires $2N-2$ bits of memory.*

Lemma 3.5 states that our algorithm has memory complexity $O(N)$, where N is the number of grid entries.

3.2. Memory Optimization

In contrast with all previous approaches presented to compute an Hilbert curve, our is the only approach that presents a linear growth of memory. The use of memory might be prohibitive for some applications or computer architectures. This led us to try to improve the amount of memory required to run Algorithm 2.

Corollary 3.5.1. *The amount of memory required to run this algorithm can be reduced to $N-2$ bits.*

Lemmas 3.1 and 3.2 ensure a bijective function exists between a primitive curve, i.e. HSFC curves with $L = 1$, and its first element. Any iteration L of a given HSFC is composed by 4^{L-1} primitive curves. If one represents all primitive curve present in a HSFC using only the first direction of each of these curves, the amount of memory needed to run this approach can be reduced to $N-2$ bits.

3.3. Generalizing the Curve

It was noticed that Algorithm 2 is quite similar to the iterative Lindenmayer-System approach. 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 the overhead of this approach in comparison to Algorithm 2, since another array must be held in memory. Ideally this array would contain $N-1$ entries and each entry would occupy exactly 1 bit.

4. Implementation Details

In this section we will briefly detail the most important implementation decisions of XOR-Hilbert. All code was written in C programming language. The alternative approaches to XOR-Hilbert are completely detailed in the main document of this thesis. In order to keep this extended abstract compact we will only detail the regular XOR-Hilbert approach.

4.1. Bit-array

As previously observed, in Lemma 3.5, 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 **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.

Algorithm 3: XOR-Hilbert

```
Input: $L$, and $d$
Output: $p$
begin
1     $p :=$ malloc(sizeof(char) * N_CHARS(L));
2     $s_1 := 192$;
3     $s_2 := 128$;
4     $s_3 := 64$;
5     $p := 27$;
6     for $l := 0;\ l < L;\ l := l + 1$ do
7         for $c := 0;\ c < 4^{l-1} - 1;\ c := c + 1$ do
8             $p[c + 4^{l-1}] := p[c] \wedge 85_{dec}$;
9             $p[c + 2(4^{l-1})] := p[c] \wedge 85_{dec}$;
10            $p[c + 3(4^{l-1})] := p[c] \wedge 170_{dec}$;
11        $\mathtt{tmp} := p[4^{l-2} - 1]$;
12        $p[4^{l-2} - 1 + 4^{l-2}] := \mathtt{tmp} \wedge 21_{dec}$;
13        $p[4^{l-2} - 1 + 2(4^{l-2})] := \mathtt{tmp} \wedge 21_{dec}$;
14        $p[4^{l-2} - 1 + 3(4^{l-2})] := \mathtt{tmp} \wedge 42_{dec}$;
15        $p[4^{l-2} - 1] := p[4^{l-2} - 1] || s_1 \wedge 64_{dec}$;
16        $p[2(4^{l-2}) - 1] := p[2(4^{l-2}) - 1] || s_2 \wedge 64_{dec}$;
17        $p[3(4^{l-2}) - 1] := p[3(4^{l-2}) - 1] || s_3 \wedge 64_{dec}$;
```

This bit array is represented by variable p, line 1 of Algorithm 3 , and has the following structure:

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

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

4.1.1 Allocation and Initialization

Allocating bit array p is quite straight forward. One only needs to know the iteration number of the desired HSFC. Following Lemma 3.5, we conclude that the amount of **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} **char** variables for the regular XOR-Hilbert version. The layout of the directions within our bit array was based on the order used by the Nano-Programs. Making it possible to extract directions from our bit array in the exact same fashion as FUR-Hilbert. The directions values, d_k, are laid out in diminishing order to k. 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. This layout results in the following scheme:

$$p = \underbrace{\{d_3,\ d_2,\ d_1,\ d_0\}}_{\mathbf{char}_0},\ \underbrace{\{d_7,\ d_6,\ d_5,\ d_4\}}_{\mathbf{char}_1}, ... \qquad (3)$$

In order to obtain direction d_k, one has to apply a shift-right operation of $2(k \bmod 4)$ bits to $p[\lfloor k/4 \rfloor]$. In order to initialize bit array p, one just needs to populate it with the directions of a primitive curve, $l = 1$, respecting the previously defined layout.

4.1.2 Computing the Next Iteration

In Section 3 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 7 to 17 of Algorithm 3. 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 **char** variable contains 4 directions, it is possible to rewrite the previous layout, Formula 3, in such a way that each cell of p is represented by a **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.

The values held by linking directions, s_x, are defined in Algorithm 2, lines 2, 3 and 4. However, since these variables are stored in the 2 most-significant bits of a given **char** variable they must be padded by 4 zeroes. This implies that an exclusive-or by 64 operation must be applied to every variable s_x in order to compute the value of these directions for the next iteration.

4.2. Computing and Storing New Quadrants

Now that the boundaries of each quadrant are known, it is possible to apply the proper bit-mask, and concatenate the newly computed quadrant entries into p. are used to perform bit-wise exclusive-or operation to the first $4^{l-2}-1$ **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, $\forall_{k\in[0,4^{l-2}-1]}$:

$$p[4^{l-2}+k] := p[k] \wedge 01010101_{bin}, \tag{4}$$
$$p[2(4^{l-2})+k] := p[k] \wedge 01010101_{bin}, \tag{5}$$
$$p[3(4^{l-2})+k] := p[k] \wedge 10101010_{bin}. \tag{6}$$

Note that:

$$01010101_{bin} = 85_{dec},$$
$$10101010_{bin} = 170_{dec}.$$

It was decided to use decimal representation within Algorithm 3 for better readability.

Literal values 01010101_{bin} and 10101010_{bin}, lines 12 to 14 of Algorithm 3, are used to compute the value of the last **char** variable, in each of the last three quadrants:

$$p[2(4^{l-2})-1] := p[4^{l-2}] \wedge 00010101_{bin}, \tag{7}$$
$$p[3(4^{l-2})-1] := p[4^{l-2}] \wedge 00010101_{bin}, \tag{8}$$
$$p[4^{l-1}-1] := p[4^{l-2}] \wedge 00101010_{bin}. \tag{9}$$

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.

5. 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 2.2, resulting in Algorithm 4.

5.1. User-Defined Functions

Our implementation only computes and processes a given HSFC, 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: Process Curve

```
Input: $P$, and $C$
begin
1     $(i, j) := (0, 0)$;
      //process_entry;
2     for $c = 0;\ c < C-1;\ c := c+1$ do
3         $p := P[c]$;
          for $k := 0;\ k < 4;\ k := k+1$ do
4             $d := p \bmod 4$;
5             $p := p >> 2$;
6             $i := (d-2) \bmod 2$;
7             $j := (d-1) \bmod 2$;
              //process_entry;
8     $p := P[C-1]$;
9     for $k := 0;\ k < 3;\ k := k+1$ do
10        $d := p \bmod 4$;
11        $p := p >> 2$;
12        $i := (d-2) \bmod 2$;
13        $j := (d-1) \bmod 2$;
          //process_entry;
```

6. 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 **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.

7. Approach Comparison

All approaches, presented in Section 2.2, 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. Adapting this loop to a stand-alone C program was fairly straight forward. One just need to follow the instructions that were made by its authors. 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 [4]. 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. Tables 1 and 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 labels will be carried out 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.

7.1. Run-Time Analysis

As it can be seen in Table 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 approaches based on XOR-Hilbert present an increase in performance, in comparison to these other two approaches. However, both the memory optimized version and the version that makes use of Nano-Programs present worse performance than XOR-Hilbert.

7.2. 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, in contrast to the approaches presented by [1], which require a constant amount of 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, which is relatively small given that N grows exponentially. The memory analysis for the alternative approaches to XOR-Hilbert are detailed in the main document of this thesis. It is possible to observe the memory requirements for XOR-Hilbert, FUR-Hilbert and the iterative version of the Lindenmayer-System with more detail in Table 2.

8. Applications Evaluation

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.

A thorough analysis of cache behaviour is present on the main document of this thesis. Once again, in order to keep this extended abstract compact we opted not to present this analysis. Due to the overhead presented by the cache-oblivious approaches, if one of these approaches presents an improved running time in comparison to the naive one it is implied that this approach improved the cache behaviour greatly.

8.1. 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.

8.1.1 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, label 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

Table 1: Run-time comparison in seconds.

Iteration	XOR	FUR	LDN
1	9.1750e-05	3.25e-06	2.75e-06
2	8.825e-05	3e-06	3.25e-06
3	8.8749e-05	3.5e-06	5.5e-06
4	9.025e-05	5.25e-06	1.6250e-05
5	9.725e-05	1.2750e-05	5.7749e-05
6	0.000119	4.3e-05	0.000225
7	0.000212	0.000163	0.000898
8	0.000585	0.000649	0.003583
9	0.002064	0.002574	0.014346
10	0.008115	0.010312	0.057507
11	0.031826	0.041166	0.230775
12	0.126627	0.167088	0.833475
13	0.507307	0.573634	2.041121
14	1.125117	1.503230	7.001975
15	3.894231	5.0928702	26.61183

Table 2: Memory comparison in bytes.

Iteration	XOR	FUR	LDN
1	1365	1364	1374
2	1368	1355	1371
3	1380	1352	1382
4	1370	1367	1389
5	1378	1397	1412
6	1365	1389	1398
7	1386	1356	1365
8	1369	1382	1341
9	1462	1384	1375
10	1657	1409	1390
11	2384	1373	1375
12	5507	1374	1370
13	17748	1350	1375
14	66957	1386	1375
15	263547	1342	1386

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.

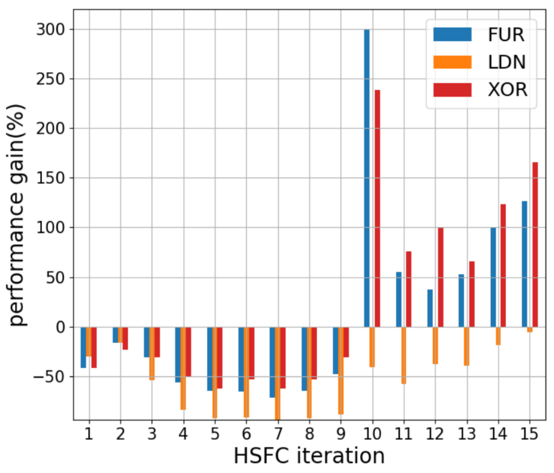

Figure 1: Matrix transposition performance gain.

8.2. Floyd-Warshall Algorithm

Another algorithm used by *Böhm et al.* [1] to evaluate the performance of this cache-oblivious loop was the Floyd-Warshall algorithm [3]. 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 $n \times n$. For improved cache behaviour the adjacency matrix was previously transposed for all approaches.

8.2.1 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 8.1, 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%.

8.3. Matrix Multiplication

In [1], one of the applications used to test the efficiency of FUR-Hilbert was the matrix multiplication [3]. 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

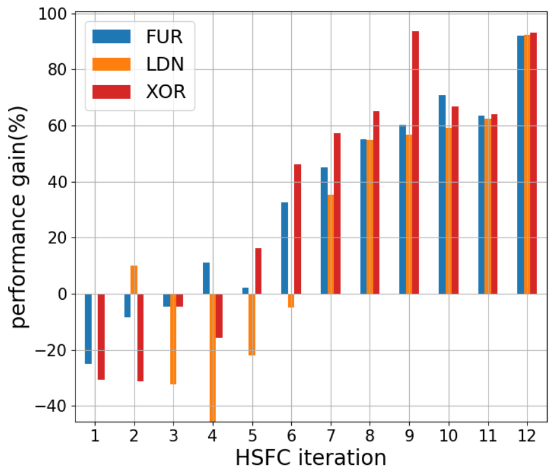

Figure 2: Floyd-Warshall algorithm performance gain.

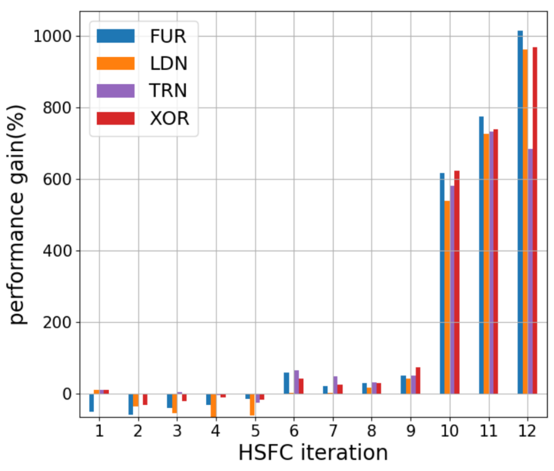

Figure 3: Matrix Multiplication performance gain.

to be stored in memory. These measurements were compared against two different naive approaches. The first approach follows the most naive version of a matrix multiplication. 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. This optimization allows matrix B to be scanned in an optimized fashion, since subsequent visited entries will be found in adjacent memory positions, in contrast with the first naive approach. This optimized approach is referenced by label TRN.

8.3.1 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 1. 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.

9. 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 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.

10. 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, like MKL-BLAS [6].

In this paper we already demonstrated that it is possible to generalize the curves produced by our approach making use of Nano-Programs, Section 3. 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.

Acknowledgements

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.

References

[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*, pages 17–26. IEEE, 2016.

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

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

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

[5] R. Goldman, S. Schaefer, and T. Ju. Turtle geometry in computer graphics and computer-aided design. *Computer-Aided Design*, 36(14):1471–1482, 2004.

[6] Intel. Blas and sparse blas routines. shorturl.at/blmzA, Sep 2019. (visited on 2019-10-5).

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

[8] P.Prusinkiewicz. L-systems: from the theory to visual models of plants. 1996.

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

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

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

[12] D. Voorhies. Space-filling curves and pace-filling curves and a measure of coherence. *Graphics Gems II*, page 26, 1991.