A Novel Hilbert Curve for Cache-locality Preserving Loops

Christian Böhm, Martin Perdacher, and Claudia Plant

Abstract—Modern microprocessors offer a rich memory hierarchy including various levels of cache and registers. Some of these memories (like main memory, L3 cache) are big but slow and shared among all cores. Others (registers, L1 cache) are fast and exclusively assigned to a single core but small. Only if the data accesses have a high locality, we can avoid excessive data transfers between the memory hierarchy. In this paper we consider fundamental algorithms like matrix multiplication, K-Means, Cholesky decomposition as well as the algorithm by Floyd and Warshall typically operating in two or three nested loops. We propose to traverse these loops whenever possible not in the canonical order but in an order defined by a space-filling curve. This traversal order dramatically improves data locality over a wide granularity allowing not only to efficiently support a cache of a single, known size (cache conscious) but also a hierarchy of various caches where the effective size available to our algorithms may even be unknown (cache oblivious). We propose a new space-filling curve called Fast Unrestricted (FUR) Hilbert with the following advantages: (1) we overcome the usual limitation to square-like grid sizes where the side-length is a power of 2 or 3. Instead, our approach allows arbitrary loop boundaries for all variables. (2) FUR-Hilbert is non-recursive with a guaranteed *constant* worst case time complexity per loop iteration (in contrast to O(log(grid-size)) for previous methods). (3) Our non-recursive approach makes the application of our cache-oblivious loops in any host algorithm as easy as conventional loops and facilitates automatic optimization by the compiler. (4) We demonstrate that crucial algorithms like Cholesky decomposition as well as the algorithm by Floyd and Warshall by can be efficiently supported. (5) Extensive experiments on runtime efficiency, cache usage and energy consumption demonstrate the profit of our approach. We believe that future compilers could translate nested loops into cache-oblivious loops either fully automatic or by a user-guided analysis of the data dependency.

Keywords—Cache-oblivious; Hilbert curve; Z-order curve;

1 Cache-oblivious Algorithms

Countless algorithms from data analysis, basic math [1], graph theory, etc. are formulated as two or three nested loops which process a larger collection of objects. Let us for instance consider the simple algorithm of matrix multiplication $A := B \cdot C$ determining the entries $a_{i,j}$ of $A \in \mathbb{R}^{n \times m}$ by the rule:

$$a_{i,j} := \sum_k b_{i,k} \cdot c_{k,j}.$$

- *C. Böhm, Ludwig-Maximilians-Universität München, Munich, Germany, boehm@ifi.lmu.de*
- *M. Perdacher, C. Plant, University of Vienna, Vienna, Austria, {martin.perdacher, claudia.plant}@univie.ac.at; C. Plant, ds:UniVie*

Since in C-like languages matrices are stored in a row-wise order, it is common practice to transpose C before computing the scalar product $\sum_k b_{i,k} \cdot c^{\mathsf{T}}_{j,k}$ to achieve a higher access locality:

for $i := 0$ **to** $n-1$ **do**
  for $j := 0$ **to** $m-1$ **do** $a_{i,j} := \sum_k b_{i,k} \cdot c^{\mathsf{T}}_{j,k}$;

This algorithm essentially reads B one time, row by row, from main memory into cache. For each row of B all the rows of C^{T} are read into cache, and combined with the current row $B_{i,*}$. Unless the complete matrix C^{T} fits into cache, this cyclic access pattern leads to a failure of the cache mechanism: with strategies like LRU (Least Recently Used), every row of C^{T} will be removed from cache before it can be re-used. As a consequence, we have a total of n transfers of the complete matrix C^{T} from main memory to cache. We could make our algorithm **cache-conscious** [2] by an additional loop:

for $I := 0$ **to** $n-1$ **stepsize** s **do**
  for $j := 0$ **to** $m-1$ **do**
    for $i := I$ **to** $I+s-1$ **do** $a_{i,j} := \sum_k b_{i,k} \cdot c^{\mathsf{T}}_{j,k}$;

Provided that we have a single cache, large enough to store s rows of B and 1 row of C^{T}, this strategy is dramatically better, because now we have to transfer C^{T} from main memory to cache only $\lceil n/s \rceil$ times while we still transfer matrix B once. Modern processors support a memory hierarchy involving 2–3 levels of cache (L1, L2, L3, ordered by decreasing speed and increasing size), as well as a set of registers which are even faster than L1 cache. The main memory is usually organized as a virtual memory. Apart from expensive swapping to hard disk or solid state disk (if the matrices B and C^{T} do not fit entirely into the physical main memory) we have to consider a second locality issue: the translation of virtual into physical addresses is supported by a very small associative cache called translation look-aside buffer. Only for a small number of pages this translation is fully efficient. While we might be able to determine the pure hardware size of all these cache mechanisms for a given hardware configuration it is difficult to know (and subject to frequent changes) how much of the various caches is available for our matrices, and not occupied e.g. by other concurrent processes or the operating system.

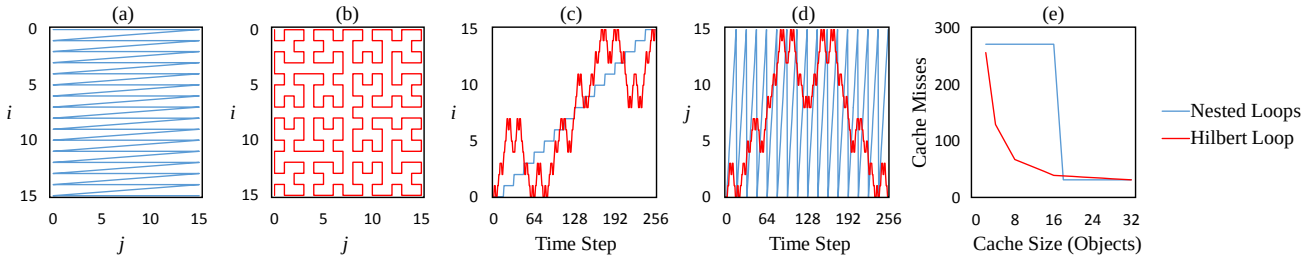

Fig. 1: Comparison of the Traversal Order for Nested Loops (a) and Hilbert Loops (b). An improved locality can be recognized in the histories over time for variable i (c) and j (d), and a considerably improved cache miss rate (e).

To efficiently support the complete hierarchy of memories of (effectively) unknown sizes, we need a different concept: a **cache-oblivious algorithm** [3] is not optimized for a single, known cache size. Its strategy supports a wide range of different cache sizes which can also be present simultaneously. The idea is to systematically interchange the increment of the variables i and j such that the locality of the accesses to both types of objects (i **and** j) is guaranteed. In Figure 1 we can recognize (a) the cyclic access pattern of nested loops, (b) the cache-oblivious access pattern of the Hilbert curve, (c) the histories of variable i and (d) j over time, and (e) the number of cache misses over varying cache size. We can see in Fig. 1(d) that the access pattern of the variable j yields much more locality for the Hilbert loops compared to the cyclic access pattern of the nested loops. The result (e) is a dramatically improved number of cache misses, particularly for realistic cache sizes like 5-20% of the main memory.

In this paper, we demonstrate how to parallelize various algorithms by assigning to each thread a contiguous part of the space-filling curve. We assume a modern multi-core processor where the shared memory is a centralized component and accesses to main memory are the main obstacle to achieve a high degree of parallelism. Improved locality of memory accesses lead to better utilization of the (in parts local) caches and increases the possible parallelism. Unlike existing work focusing on parallelism in a distributed environment (Hadoop, Spark, etc.), we focus on parallelism within a single multi-core computing nodes. Combining both levels of parallelism is an interesting topic for future work.

Contributions

We propose to replace nested loops enumerating pairs of (i, j) in canonical order by **cache-oblivious loops** following a space-filling curve. Well-known approaches like Hilbert-, Z-order-, or Peano-curve are too inefficient and limited for a loop iteration, as discussed in Section 2.1 and 2.2. Our approach called **Fast Unrestricted (FUR) Hilbert Loop** overcomes these problems:

1. We reduce the worst-case complexity per loop iteration from $O(\log n)$ to a constant complexity by making an algorithm based on a context-free grammar (Lindenmayer-System) non-recursive (cf. Section 3).
2. We overcome the usual limitation of space-filling curves to grids of equal size lengths $n \times n$ where n is a power of two or three by a concept we call *nano-programs*. Nano-programs completely avoid any additional overhead (like discarding unnecessarily generated pairs) and instead even accelerate the loop generation, cf. Section 4.
3. We implement the FUR-Hilbert Loop as a preprocessor macro which makes it extremely convenient to be used as a building block in any host algorithm and facilitates compiler optimization, cf. Section 5.
4. We show that even algorithms, which operate only on the lower or upper tridiagonal like Cholesky decomposition profit a lot by our approach. Additionally we compare the canonical version of the all-pairs shortest path algorithm by Warshall with our Hilbert approach.
5. We present extensive experiments on runtime efficiency, cache usage and energy consumption. We demonstrate that our algorithms have a small cache usage footprint and are energy efficient. Our algorithms are highly competitive to state of the art techniques.

The initial conference version of this paper introduced the FUR-Hilbert Loop [4] as a cache-oblivious approach for matrix multiplication and K-means clustering. In this paper, we extend this work in the way described by the final two contributions above and odd-size nano-programs (cf. Section 4).

We demonstrate the superiority by applying FUR-Hilbert Loops in four host algorithms: matrix multiplication, K-means clustering, Cholesky decomposition and the algorithm by Warhshall cf. Section 6. Motivated by the results of extensive experiments (cf. Section 7), we believe that future compilers could even integrate our concept and transform conventional loops into FUR-Hilbert Loops based on an automatic or user-guided analysis of the data dependency.

2 Well-known Methods for Hilbert

To facilitate later a rigorous mathematical treatment we introduce the well-known approaches in this section following an automaton-theoretic point of view. Many iterative approaches to generate space-filling curves like [5] can be regarded as a deterministic finite automaton,

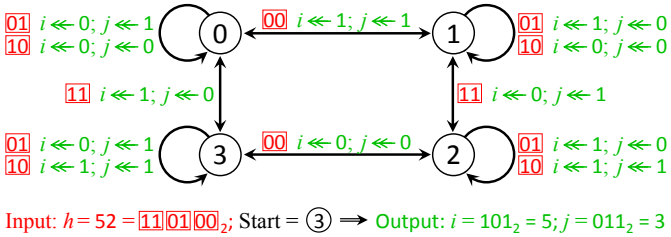

Fig. 2: Mealy-DFA for Inverse Hilbert: $(i,j) := \mathcal{H}^{-1}(h)$ to generate variables i and j from the Hilbert value h.

and many recursive approaches like [6] as a context-free grammar.

2.1 Explicit Enumeration of Hilbert Values

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

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

For matrix multiplication, we substitute our placeholder:

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

The inverse Hilbert function is depicted in Figure 2 as a deterministic finite automaton (DFA) of Mealy-type (i.e. the output, the bit-strings representing i and j are generated at the state transitions of the DFA). The bit-string representation of the Hilbert value h, divided into groups of 2 bits, is the input of the DFA. State 3 is used as starting state, to generate the Hilbert curve in a clockwise order. Alternatively, State 2 can also be used for start to generate an anti-clockwise curve. Appending a binary digit 1 to the output bit-string i (in symbols: $i \twoheadleftarrow 1$) represents the mathematical operation $i := 2 \cdot i + 1$. With the example input $h = 52_{10} = 110100_2$, our DFA makes upon the first bit-pair 11_2 of the input string the first transition from State 3 to State 0, then for the second bit-pair 01_2 stays in State 0 and goes finally with bit-pair 00 to State 1. In these three state transitions it appends 1, 0, and 1 to i; 0, 1, and 1 to j to finally obtain $(i,j) = (5,3)$ in decimal system. Each of the $\log_2 n$ bit-pairs of h is separately processed by the DFA. In contrast to this $O(\log n)$ overhead, a pair of two nested loops (cf. Section 1) has only a constant overhead per loop iteration (increment i and/or j). The overhead could be prohibitive for many applications. When the upper limits of the loops are different ($0 \le i < n, 0 \le j < m$) or do not correspond to a power of two then we have two options: either we generate a bigger Hilbert curve and suppress the processing of pairs (i,j) that are actually out of the range:

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

or we generate a Hilbert curve of side length $2^{\lfloor \log_2 \min\{n,m\} \rfloor}$ and complement the missing values later in additional loops with smaller Hilbert curves [7]. Both options incur a high overhead and deteriorate the goal of cache-obliviousness.

2.2 Lindenmayer-Systems

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

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

$$\begin{array}{lcl|l} A & \to & \pi & \ominus\ B \triangleright \oplus\ A \triangleright A \oplus \triangleright\ B \ominus \\ B & \to & \pi & \oplus\ A \triangleright \ominus\ B \triangleright B \ominus \triangleright\ A \oplus \end{array}$$

The terminal symbols represent the graphical operations:

- $\ominus$ *turn 90 degrees to the left without moving,*
- $\oplus$ *turn 90 degrees to the right without moving,*
- $\triangleright$ *go forward one step in the current direction* d*,*
- π *process object pair* (i,j)*,*

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

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

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

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

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

While the Mealy-DFA of Section 2.1 generates only one (i,j)-pair, the Lindenmayer-CFG produces the whole $n \times n$ Hilbert curve when we start at level $\ell = \log_2 n$. Axiom A with $d = 3$ generates the values in a clockwise order starting from $(i,j) = (0,0)$ and ending at $(n,0)$, and axiom B with $d = 2$ in an anticlockwise order ending at $(0,n)$.

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

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

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

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

The **mod**-operation preserves the sign. For $d = \langle 0, ..., 3 \rangle$ we get $i := i + \langle 0, -1, 0, +1 \rangle$ and $j := j + \langle -1, 0, +1, 0 \rangle$.

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

Algorithm 1 Recursive Lindenmayer Algorithm.

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

Analogously **function** $B(\ell)$; the labels are not needed for the implementation but for the following analysis. Note the comments giving the corresponding terminal and nonterminal symbols from the context-free grammar. Although all the increase and decrease operations corresponding to the commands $\triangleright, \oplus$, and $\ominus$ have now constant complexity, the recursive implementation still has some drawbacks: Firstly, after the generation of 4 subsequent (i, j)-pairs, one incarnation of $A(\ell)$ or $B(\ell)$ is finished. We have to return to one of $\text{labelA}_{0..3}$ or $\text{labelB}_{0..3}$, perform there the next actions and then start the next recursive calls. In summary, after every 4^k iterations we have to move up and down on the stack at least for k positions where $1 \le k \le \log_2 n$. This is still a logarithmic overhead per loop iteration in a worst-case-analysis (but since the worst case does not occur frequently it is indeed constant in the average-case when applying amortized analysis). Secondly, the problem of grids with different side lengths not corresponding to powers of two is still open in the recursive solution.

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

3 Novel Non-recursive Lindenmayer

The functions $A(\ell)$ and $B(\ell)$ are not straightforward to make iterative. In regular time intervals, we have to leave one or more recursive incarnations of $A(\ell)$ or $B(\ell)$, return to the middle of another A or B-function on the recursion stack, perform the next action of this incarnation then, and start again new recursive calls. This can be studied in Figure 3, where we are in the middle of the generation of the Hilbert loop, at $i = 5, j = 3, h = 52$. At this point we have four active incarnations, one of production rule A, two of B, as well as the terminating rule $A \to \pi$. The dark blue printed rule at $\ell = 3 = \log_2 n$ generates the Hilbert curve of the whole (i, j)-grid (surrounded by a dark blue frame). We are currently in the last (lower left) sub-quadrant, marked by a light blue frame, and the corresponding position in production rule A is labelA_3 which is expanded in the next, light blue production rule. In a recursive implementation, we have labelA_3 on the stack. The production rule $B(\ell = 2)$ in turn is expanded at labelB_1 (next position on the recursion stack) corresponding to the green production rule (again B) and the green frame in the grid. When ending the green production rule, we have to return to labelB_1 of the light blue rule ($\ell = 2$), perform the corresponding action ($\triangleright$, between labelB_1 and labelB_2), and then make the next recursive call $B(\ell-1)$ at labelB_2. The labels which are on the stack, $\text{labelA}_{0..3}$, $\text{labelB}_{0..3}$ agree with the definitions in **function** $A(\ell)$ and $B(\ell)$. Where appropriate, we will also note e.g. labelX_0, meaning: labelA_0 *or* labelB_0.

Our idea to make the recursive algorithm iterative is to code the complete recursion stack in a single $\mathbb{N}_0$-variable (e.g. 64 bit, possibly a register of the CPU). We will demonstrate that we do not need a new variable for this purpose. Instead, the current Hilbert value h as well as the current direction code d contain all the information to derive both, the current recursion depth,

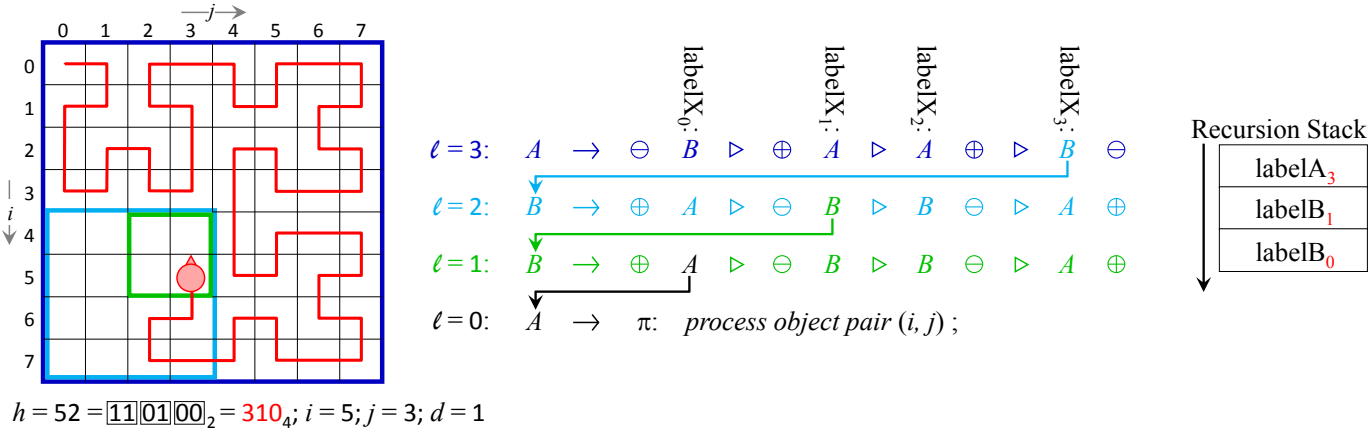

Fig. 3: Recursive Generation of (i, j)-pairs Following the Hilbert-curve.

as well as (for all recursive incarnations of the methods A and B) the current position where we have to return. We split the proof of this in two parts: Lemma 1 states that the number of each label on the recursion stack exactly corresponds to the Hilbert-value grouped in bit-pairs. This is already obvious from Figure 3 where the Hilbert value $h = 52 = 11\,01\,00_2 = 310_4$ noted in the 4-adic system is identical to the numbers on the stack: (labelA$_3$, labelB$_1$, labelB$_0$). Lemma 2 shows how to derive the information whether we are in $A(\ell)$ or $B(\ell)$.

Lemma 1 (Label-Number)**.**
Consider an incarnation of the non-terminating rules:

$$A \;\to\; \ominus\ B\ \triangleright\ \oplus\ A\ \triangleright\ A\ \oplus\ \triangleright\ B\ \ominus \quad \textit{or}$$
$$B \;\to\; \oplus\ A\ \triangleright\ \ominus\ B\ \triangleright\ B\ \ominus\ \triangleright\ A\ \oplus \quad \textit{at level } \ell.$$

(1) *The number pop(ℓ) of processed object pairs (i, j) starting from the incarnation at level ℓ is pop$(\ell) = 4^\ell$.*
(2) *The incarnation overall increases h by forw$(\ell) = 4^\ell - 1$.*
(3) *For each Hilbert value h generated at* labelX$_k$*, the value*

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

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

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

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

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

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

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

Lemma 2 (Production Rule A or B)**.**
(1) *The direction code d is the same at the beginning and at the end of a production rule.*
(2) *At the beginning and end of the production rule A, the parity of the direction code d is always odd, at the beginning and end of B always even.*
(3) *The parity of d combined with the position (e.g. identified by the label) decides if we are in grammar rule A or B.*

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

$$A \;\to\; \ominus\ B\ \triangleright\ \oplus\ A\ \triangleright\ A\ \oplus\ \triangleright\ B\ \ominus$$

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

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

$$A \;\to\; \ominus\ \underset{\text{labelA}_0\downarrow}{B}\ \triangleright\ \oplus\ \underset{\text{labelA}_1\downarrow}{A}\ \triangleright\ \underset{\text{labelA}_2\downarrow}{A}\ \oplus\ \triangleright\ \underset{\text{labelA}_3\downarrow}{B}\ \ominus$$

Before expanding B at labelA$_0$ we have applied $\ominus$. Note that both operators $\oplus$ and $\ominus$ toggle the parity of d from even to odd and vice versa, because they add an odd

number (1 or 3) to d (**mod** 4). Thus, if d is odd at the start of expansion A (induction hypothesis), it is even at the start of expansion B, odd again at A at label$A_{1..2}$, and even at B at labelA_3. Analogously in the expansion of B: if d is even at the beginning (induction hypothesis), d is odd at every nonterminal A and even at every B. This is a complete analysis of all cases, and to summarize the induction step: if (2) is true for the nonterminal on the left side of a rule (at level ℓ), it must be true for all nonterminals on the right side and thus for all left sides of the next level ($\ell-1$).

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

$$\begin{array}{lccccccccccc} A \rightarrow & & \ominus & B & \triangleright & \oplus & A & \triangleright & A & \oplus & \triangleright & B & \ominus \\ & \text{odd} & | & \text{even} & | & & \text{odd} & | & & \text{even} & | & & \text{odd} \end{array}$$

$$\begin{array}{lccccccccccc} B \rightarrow & & \oplus & A & \triangleright & \ominus & B & \triangleright & B & \ominus & \triangleright & A & \oplus \\ & \text{even} & | & \text{odd} & | & & \text{even} & | & & \text{odd} & | & & \text{even} \end{array}$$

We can see that the parity differs between A and B at every position. Therefore, the combination of position and parity decides the rule A or B in which we currently are. □

Now, Lemma 1 and 2 would enable us to exactly mimic the recursion with its stack. However, we would still have the logarithmic worst-case complexity. Instead, we further transform our algorithm into that in Figure 2 which truly performs a loop enumerating the Hilbert values h. In its body, we perform the operation π (process (i,j)) and, then decide where we would be in the recursive system and which action a must be executed. An action takes always place between two successive recursive calls and has the label of the latter (e.g. $a=2$ is between labelX_1 and labelX_2), and corresponds always to a forward-step, potentially preceded or followed by a $\oplus$ or $\ominus$-step. E.g. in production rule A at $a=1$ we perform $\oplus\triangleright$, and at $a=3$ we perform $\triangleright\oplus$ etc. As there is no forward-step before labelX_0 and after labelX_3 we only consider the three actions $a\in\{1,2,3\}$, but not 0. We will see later how to cope with the $\oplus$ and $\ominus$ at the beginning and end of the rules.

To obtain the right action code, we first increase h.

Algorithm 2 The Non-recursive Lindenmayer Alg.

```
1  function LindenmayerNonRecursive()
2      (i, j) := (0, 0); h := 0; d := 3;
3      while h < n^2 do
4          process object pair (i, j);
5          h := h + 1;
6          ℓ := ⌊½ log2(h and_bitw −h)⌋ + 1;
7          a := ⌊h/4^(ℓ−1)⌋ mod 4;
8          d := d xor_bitw (11_2 · (isOdd(ℓ−1) xor a = 3));
9          j := j + ((d − 1) mod 2);
10         i := i + ((d − 2) mod 2);
11         d := d xor_bitw (isOdd(ℓ−1) xor a = 1);
```

If we are at the end of one or more production rules on the stack, then increasing h will change one or more 11_2-bit-pairs at the end of h into 00_2. (cf. Lemma 1). In this case, the first bit-pair (from right to left) $\neq 00_2$ defines the level ℓ and the action a to be performed. Therefore, we determine (after increasing h) the number of 00_2-bit-pairs at the end in constant time by the following trick: If the bitwise and-operation is applied to h and its negative complement $-h$, then in the result all bits are 0 up to one exception: The last (least significant) 1 which has been set in h is still set (the result is the largest power of two which divides h with no rest). The binary logarithm $\lfloor\frac{1}{2}\log_2(h\ \mathbf{and}_{\text{bitw}}\ -h)\rfloor$ corresponds to the number of zero-pairs at the end of h and equals $\ell-1$. We determine the binary logarithm by casting the result of the bitwise **and** to a double-precision floating point number and extracting the exponent, which is very efficient and works for $1\le h\le 2^{52}-1$, the greatest natural number that can be represented by double-precision floating point numbers (according to IEEE-754) at no loss of precision. The action code then is extracted from h using

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

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

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

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

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

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

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

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

4 Nano-Programs

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

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

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

A nano-program is a bit-sequence which can be stored in an integer variable P which may be assigned to a register by the compiler while working with it. The bit-sequence contains codes similar to the direction-codes stored in the variable d (cf. Section 2.2). The nano-programs read from right to left, and therefore, the first operation to be performed on the variables i and j can be extracted from P and stored into a temporary variable c by $c := P$ **mod** 4. The current operation is then executed (i and j are updated based on c as in Section 2.2) and removed from P by a right-shift $P := \lfloor P/4 \rfloor$ until the nano-program is empty. To each sub-grid cell of size $r \times s$ there belongs a nano-program of size $r \cdot s - 1$ digits from a 4-ary system, i.e. $(r \cdot s - 1) \cdot 2$ bit. We have separate nano-programs for every size of the basis-grid $\{0,1,2,3,4\} \times \{0,1,2,3,4\}$ and for every orientation defined by the variable d, totalling in a number of $5 \cdot 5 \cdot 4 = 100$ nano-programs of sizes up to 30 bit (integer register). A few examples of nano-programs (all for orientation $d = 2$; patterns for other orientations obtained by rotation) are visualized in Figure 4: we can see the graphical access patterns for $2 \times 2 ... 4 \times 4$ grids and the corresponding nano-programs. Note that these programs read from right to left, and they are here noted in a 4-ary system. The first movement of the $2{\times}2$ pattern (step down) is coded by the tailing digit 3 of the nano-program $123_4 = 27_{10}$. The pseudo-code is embedded in the algorithm LindenmayerNonRecursive() and all processing of nano-programs highlighted in Algorithm 3. The size $(r \times s)$ of each cell of the nano-program is determined in Line 3 such that the sub-grid heights $r = 2, 3,$ and 4 are evenly distributed to sum up to n (analogously for the widths s). After a complete nano-program consisting of $r \cdot s - 1$ steps has been processed, a final movement (lines 11–16) is performed to connect the previous sub-grid cell to the subsequent one.

Algorithm 3 Lindenmayer with Nano-programs.

```
1  (i,j) := (I,J) := (0,0); h := 0; d := 3; t := ⌊log2 n⌋;
2  while h < 2^(2t−2) do
3      r := ⌊((I+1)·n)/t⌋ − ⌊(I·n)/t⌋; s := ⌊((J+1)·m)/t⌋ − ⌊(J·m)/t⌋;
4      P := nanoprograms[r][s][d];
5      while P ≠ 0 do
6          process object pair (i,j);
7          c := P mod 4;
8          P := ⌊P/4⌋;
9          j := j + ((c − 1) mod 2);
10         i := i − ((2 − c) mod 2);
11     h := h + 1;                                  } as in Alg. 2, Line 5–10
...    ...
16     i := i + ((d − 2) mod 2);                    }
17     J := J + ((d − 1) mod 2);
18     I := I + ((d − 2) mod 2);
19     d := d xor_bitw (isOdd(ℓ − 1) xor a = 1);
```

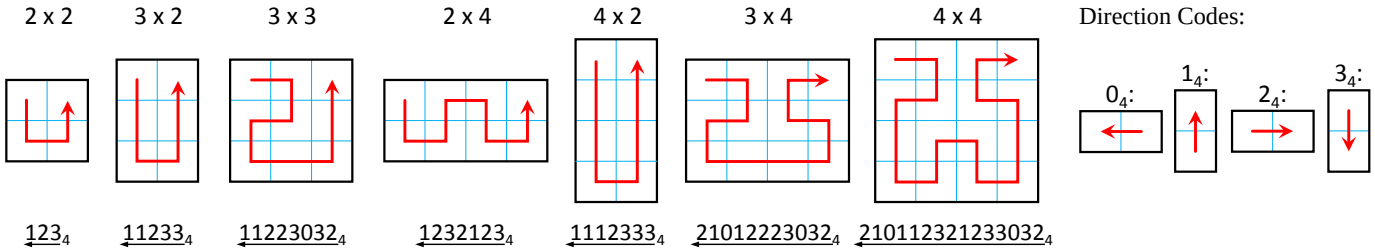

Fig. 4: Examples of Nano-programs for Grids Ranging from 2×2 to 4×4 (all having basic orientation $d = 2$).

Odd-sized Cells of Nano-programs

As depicted in Figure 4, all nano-programs for $2 \times 2 ... 4 \times 4$ grids exist. However, for the non-square sub-grids ($2 \times 3, 2 \times 4$, and 3×4) we actually need two versions: those beginning and ending at the longer side and those beginning and ending at the shorter side. For the 2×4 nano-programs, both versions exist: in addition to the nano-program 1232123_4 beginning and ending at the longer side, we can also define the 4×2 nano-program 1112333_4 (read from right to left cf. Fig. 4) beginning and ending at the shorter side. In contrast, a 2×3 sub-grid beginning and ending at the longer side would require a diagonal transition (being less local and requiring more than two bits).

Although odd-sized nano-programs are necessary if we want to support a global grid size with one or both side lengths being odd we can completely avoid the non-existing nano-programs if we carefully plan where to place the $3 \times \{2, 3, 4\}$ nano-programs. Our key idea is, at most places to allow only even-sized nano-programs. Every grid where both side lengths are even can be completely tesselated with only even-sized nano-programs. If one side length of the grid is even, and the other is odd, we can place all 3×2 and 3×4 nano-programs at that side of the grid opposite to the starting and ending point, as depicted in Figure 5. We have to control the traversal order of the grid (by specifying the initial direction $d = 3$ or $d = 2$) to make sure that the even-sized side length is opposite to the global starting and ending point (Figure 5, middle).

As depicted in Figure 5, right side, the situation is more tricky if both side lengths are odd. There, we have to place a column of 3×2 nano-programs at the right side and a single 3×3 nano-program at either of the ends of this column (in Figure 5, on the upper right corner). The Hilbert-subcurves at the upper and lower row are, unfortunately, not oriented such that the remaining 3×2 nano-programs can be positioned in a single row. The drawn layout with sub-curves oriented towards the middle is generated by bitwise logic operations.

Severely Asymmetric Grids

If $\lfloor \log_2 n \rfloor < \lfloor \log_2 m \rfloor$ like in Figure 6 where $\lfloor \log_2 n \rfloor = 2$ and $\lfloor \log_2 m \rfloor = 3$ we put a number $m' = \lceil m/2^{\lfloor \log_2 n \rfloor} \rceil$ of independent curves side by side, where the first has width $m - \lfloor \log_2 n \rfloor \cdot (m' - 1)$ and the remaining have width $\lfloor \log_2 n \rfloor$. All the curves are anticlockwise (initial $d = 2$). With this setting it is guaranteed that the nano-programs of size $3 \times \{2, 3, 4\}$ can be placed exactly such that no diagonal transitions are needed. If $\lfloor \log_2 m \rfloor > \lfloor \log_2 n \rfloor$ we analogously put $n' = \lceil n/2^{\lfloor \log_2 m \rfloor} \rceil$ clockwise curves (initial $d = 3$) one above the other. Finally we note that starting with lower bounds for (i, j) different from $(0, 0)$ is also straightforward possible (cf. Figure 6 where we start at $(i, j) = (2, 0)$) at no extra cost per loop iteration. Degenerate loops where one or both variables iterate over one value only or no values at all (empty loops) are also considered, but these extensions are left out in our pseudo-code for clarity and space restrictions.

5 Overall Architecture

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

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

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

#define FurHilbertFor $(i, j, imin, imax, jmin, jmax)$
#define FurHilbertEnd (i, j)

of which the first contains Alg. 3, Line 1–5, the part of the code before the placeholder *process object pair* (i, j), and the second contains Line 7–19, including all extensions described in Section 4. The C++-file can be downloaded from https://informatik.univie.ac.at/dm/downloads/.

The application of these preprocessor macros is almost as convenient as the application of standard loops. FUR-Hilbert loops can be nested with other loops and with

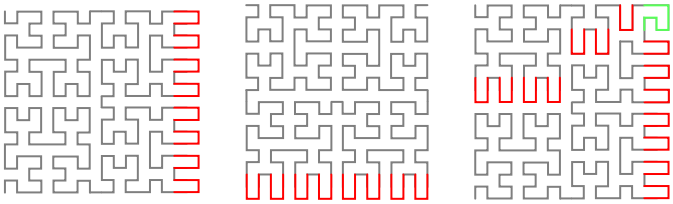

Fig. 5: Placement of 3×2 (red) and 3×3 (green) Grids.

each other. The implementor of the host algorithm can choose freely the name and type of the iterator variables (for all types allowing to apply the operator "+") and parenthesize the loop body as usual with "{" and "}". In this case editors automatically apply appropriate indentation. The following lines generate the FUR-Hilbert curve in Figure 6:

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

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

6 Cache-oblivious Loops: Applications

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

6.1 Matrix Multiplication

The matrix multiplication is vastly used in software implementations and has numerous applications [8]. The basic algorithm from our introduction to multiply $B \in \mathbb{R}^{n\times p}$ and $C \in \mathbb{R}^{p\times m}$ (stored as $C^\intercal$, additionally each row aligned to cache lines) can be straightforward implemented using a FUR-Hilbert loop, because it has no general data dependencies. If p is too large (for a cache-size of 32K say $p \gg s := 1024$), that at least a few of the rows of B and $C^\intercal$ fit into L1 cache, it is necessary to decompose the matrices horizontally into groups of s (divisible by 4 to ensure alignment with cache lines) columns before applying the FUR-Hilbert loop:

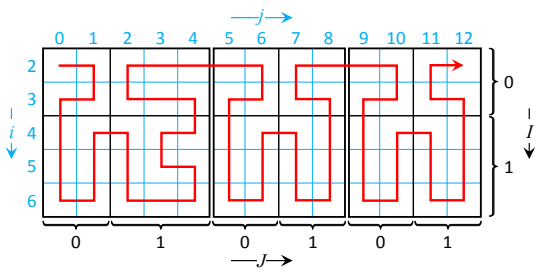

Fig. 6: **FurHilbertFor** $(i, j) \in \{2, ..., 6\} \times \{0, ..., 12\}$.

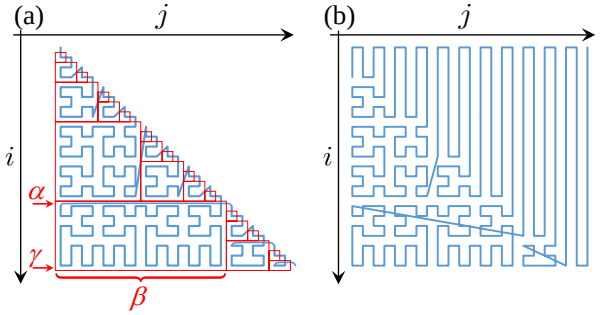

Fig. 7: Traversal of Cholesky (a) and Floyd/Warshall (b)

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

6.2 K-means Clustering

K-means, the most popular clustering algorithm, is like the related EM and K-medoid methods implemented in a loop until convergence alternately performing assignment and update steps. The expensive assignment step determines for each point x_i the distance to each cluster representative μ_j $(0 \le j < k)$ and assigns it to that cluster ID having minimal distance. We must keep track of the winner distance and the corresponding cluster ID for each point. This can be facilitated in a SIMD-parallel way by backpacking [9] the cluster ID in the least significant bits of the distance, noted $\langle dist, cID\rangle$:

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

6.3 Cholesky Decomposition

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

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

In the computation of element $\ell_{i,j}$ we read the whole matrix row $\ell_{j,*}$. This data dependency must be considered when changing the loop order of (i, j) either by space-filling curves or by parallelism. We achieve this by decomposing the lower left triangle matrix L into square blocks of a side length β being a power of two. The largest block of side length $\beta = 2^{\lfloor \log_2(n-1)\rfloor}$ is placed in the lower left corner of L starting from

row $i_{\min} = \beta$ (cf. Figure 7(a); if the overall dimension n of the matrix is not a power of two, then the blocks at the bottom lines are suitably trimmed and become a non-square $\beta \times \gamma$ rectangle). Recursively, to each placed square of side length β we connect one with side length $\frac{\beta}{2}$ on the upper left corner and on the lower right corner until the triangle is completely tesselated when we reach at β=1 or at some other defined basic resolution (for SIMD parallelism using AVX 1 or 2 we use 4×4 squares as basic elements, which degenerate to triangles at the diagonal of the matrix). Inside each block it is guaranteed that the data dependency plays no role since always $i_{\min} > j_{\max}$. We can apply FUR-Hilbert loops as well as SIMD and MIMD parallelism inside such a block but have to make sure that the blocks are ordered sequentially top-down and within the same row from left to right. The recursive block-generation is made iterative in the following pseudo-code which also considers the case of a matrix dimension n being not equal to a power of two:

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

6.4 The Algorithms by Floyd and Warshall

The Algorithm by Warshall [10] finds connected components (the transitive closure of the boolean adjacency matrix $A \in \mathbb{B}^{n\times n}$) in a directed or undirected graph. The standard algorithm uses three nested loops, resulting in an $O(n^3)$ algorithm:

for $i := 0$ **to** $n-1$ **do**
  for $j := 0$ **to** $n-1$ **do if** $a_{j,i}$ **then**
    for $k := 0$ **to** $n-1$ **do if** $a_{i,k}$ **then** $a_{j,k} :=$ **true**;

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

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

The overall traversal scheme of the two outer loops (i and j) is depicted in Figure 7(b). The algorithm by Floyd operates on a weighted matrix $A \in \mathbb{R}^{n\times n}$ and uses the same algorithmic pattern like Warshall. The line:

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

is replaced by:

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

7 Experimental Evaluation

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

7.1 Matrix Multiplication

Our algorithm FUR-Hilbert, as discussed in Section 5, is implemented in C++ and compiled with gcc version 4.9.2. We compare our algorithm to the algorithm "TifaMMy" for matrix multiplication based on the Peano Curve introduced by Bader et al. [11], [12] (source code has been obtained by the authors compiled with icc version 16.0.3). Furthermore we compare our algorithm to the specifically for Intel processors hardware- and hand-optimized Intel MKL library (https://software.intel.com/en-us/intel-mkl) version 11.3 (operation: dgemm). Popular frameworks like Apache Spark, Pythons NumPy and SciPy can be accelerated by Intel MKL. As a baseline, we also compare to the classical matrix multiplication coded in c++ transposing

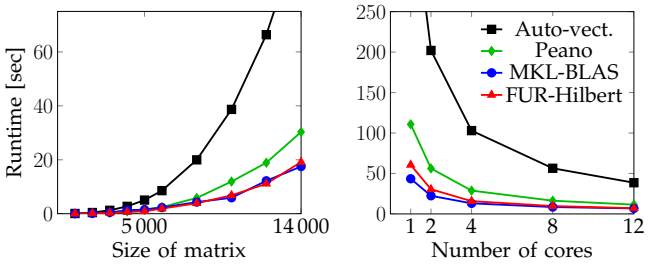

Fig. 8: Experiments on Matrix Multiplication.

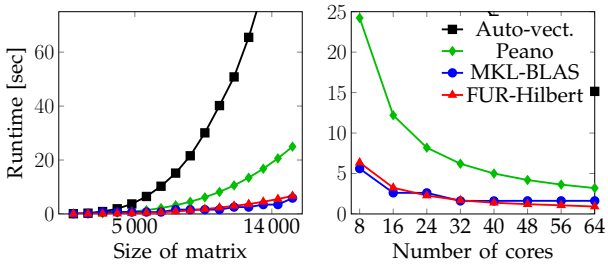

Fig. 9: Experiments on Matrix Multiplication for a Manycore System.

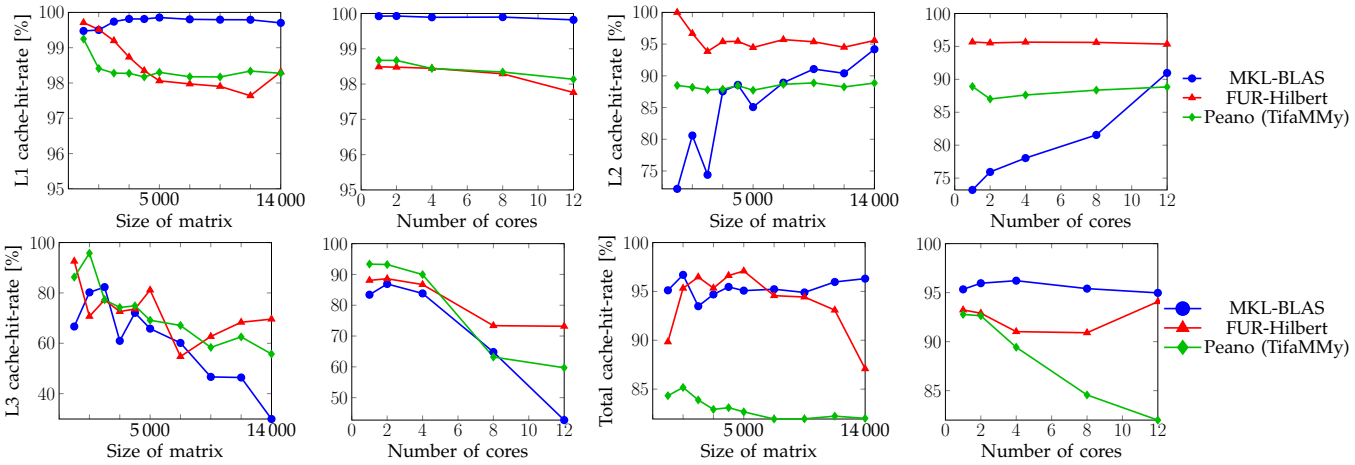

Fig. 10: Cache-hit-rate on Different Cache Levels for Matrix Multiplication. We show the cache access pattern for various threads and problem sizes. Level 1 cache-hit-rate on the top left, level 2 on the top right. At the bottom left we have the level 3 cache hit rate and on the right the cache-hit-rate among all hierarchies.

the input matrix for improved cache locality. The code has been auto-vectorized using the GNU C++ compiler.

Figure 8 (left) displays the runtime in seconds varying the size of the input matrices from 1 000 to 14 000. For larger problem sizes, the highly optimized MKL library slightly outperforms FUR-Hilbert. It processes the largest matrix in 17.50 seconds. Our algorithm needs 19.15 seconds and is at least 30% faster than the approach by Bader et al. (Peano). Peano requires 13.15 seconds. Our approach is more than 10 times faster than auto-vectorization which needs 1.7 minutes. Figure 8 (right) displays th runtime varying the number of threads. Two matrices with 10 000 elements each are processed by all techniques. All methods profit a lot from multi-threading. Auto-vectorization and Peano show almost linear speed-up albeit at a low level of overall performance. FUR-Hilbert shows similar speed-up characteristics as MKL-BLAS.

7.2 Matrix Multiplication on a Manycore-System

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

7.3 Cache hierachy on Matrix Multiplication

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

Figure 10 illustrates the cache hit rate for each cache level respectively and the cache hit rate for the entire cache hierachy. For the L1 cache-hit-rate (top left) the hardware optimized MKL-BLAS is the most cache-efficient algorithm. Our algorithm performs well for small sizes and as well as Peano for larger sizes. All of the algorithms reached at least 97.5% of the L1 cache hit rate. For the L2 cache hit rate our algorithm performs best and remains within a lower bound of 94% whereas Peano and MKL-BLAS have lower bounds of 87% and 72%. The last level cache (LLC), depicted as level 3 cache at the bottom left in Figure 10, is the slowest but largest in the cache hierarchy. Our algorithm shows good performance and uses most of the LLC for large matrix sizes and for the maximum number of threads, whereas

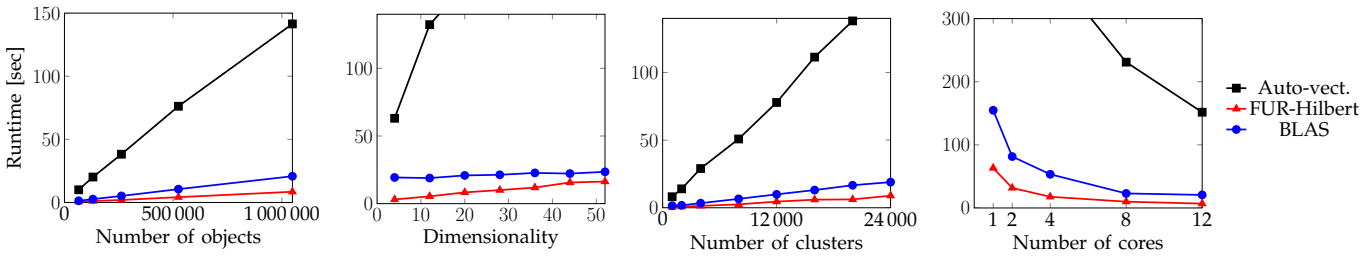

Fig. 11: Experiments on K-means Clustering.

MKL-BLAS drops down in cache hit rates. The Peano algorithm performs relatively equal to our algorithm, but shows performance decrease for large problem sizes.

We also captured the access pattern of memory access which could be served by any of the cache levels in Figure 10 (bottom right). For the total cache hit rate our algorithm competes with MKL-BLAS for sizes between 2 000 and 12 000 but drops down at both tails for matrix sizes of 1 000 and 14 000. Our algorithm clearly outperforms the Peano algorithm in both, variation of matrix sizes and thread counts.

7.4 K-means

Here, we extended our K-means implementation [9] with the Hilbert curve. We use the same comparison methods as for matrix multiplication but exclude the Peano-curve based algorithm by Bader et al. [11], [13] since this approach is not designed to support K-means and is outperformed by MKL-BLAS on the task of matrix multiplication. The MKL library also does not include K-means, however it can be used to speed up the distance calculations. The MKL-based technique computes the scalar products between objects and cluster centers by matrix multiplication. As the runtime of K-means strongly depends on the number of iterations, we compare it for a fixed number of 5 iterations. As a basic setting, we consider a 20-dimensional data set with 1 048 576 data objects and 24 000 clusters. A high number of clusters causes massive runtime but is practically relevant, e.g., for the coarse quantization step of the product quantization indexing technique [14].

Figure 11 (left) displays the runtime when varying the number of objects. Our technique processes 1 million objects in less than 8 seconds while auto-vectorization needs 2.88 minutes for 5 iterations of K-means. The next sub-figure varies the number of dimensions. The rightmost sub-figures display the speed-up which is similar to matrix multiplication.

We have also tested the hardware-optimized library DAAL (Intel Data Analytic Acceleration Library, https://software.intel.com/en-us/intel-daal), but DAAL does not compete in this setting. DAAL is a library of optimized algorithmic building blocks for data analysis and solves problems which are associated with "Big Data". This includs regression, classification or related problems as well as clustering problems like K-means. We have been using DAAL with the current version of Intel Parallel Studio XE (version 2017 update 3). Unfortunately DAAL cannot handle settings with a high number of clusters, where $K > 2\,050$. Even for $K = 1\,000$ or $K = 2\,000$ it performs worse than our auto-vectorization approach and needs 49.42 and 97.74 seconds for completion. We had been running DAAL in the batch processing mode, with the same settings which have been used for our algorithms.

7.5 Cholesky Decomposition

All implementations of the Cholesky decomposition take a positive-definite matrix A and apply the decomposition of the form $A = LL^T$ in double precision. Our algorithm is implemented using C++ (compiled with gcc version 6.4.0 and OpenMp 4.5). We compare our algorihtm to the hardware optimized library LAPACK implemented in the Intel Math Kernel Library (MKL version 17.0 update 4, see https://software.intel.com/en-us/mkl/features/linear-algebra) and to the Parallel Linear Algebra Software for Multicore Architectures (PLASMA version 2.8.0) library https://bitbucket.org/icl/plasma. In contrast to LAPACK, which relies on BLAS level 2 calls, PLASMA uses BLAS level 3 calls, especially suitable for algorithms like Cholesky decomposition [15]. As a baseline we also compare our algorithm to a classical implementation of the Cholesky decomposition, which has been automatically vectorized using the gnu gcc compiler (Auto-vect.).

Figure 12 (left) displays the runtime in seconds on different dimensions of the input matrix varying from 1 000 to 12 000. For the largest matrix of 12 000, the hardware optimized MKL-LAPACK algorithm shows the best performance with 2.04 seconds whereas our algorithm FUR-Hilbert needs 3.66 seconds. However, our algorithm

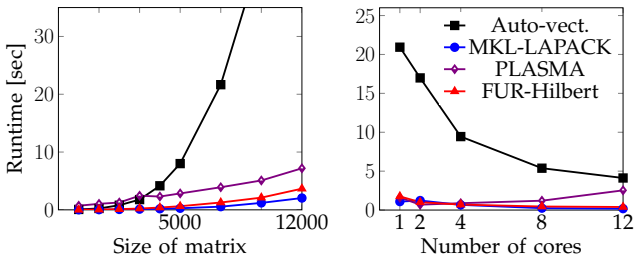

Fig. 12: Experiments on Cholesky Decomposition.

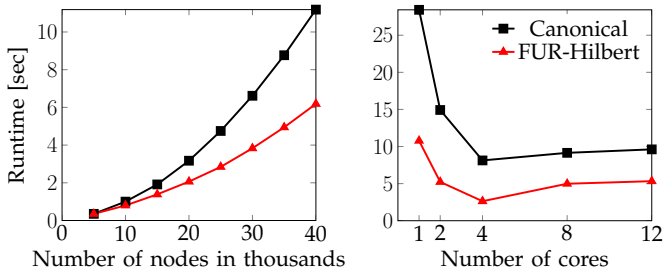

Fig. 13: Experiments on the Algorithm by Warshall.

is at least 20% faster than the PLASMA library, which needs 7.18 seconds and 17 times faster than the auto-vectorization taking 123.64 seconds to completion.

Figure 12 (right) displays the runtime varying the number of threads. Each algorithm processes a matrix of size 5 000. All algorithms profit from multithreading. Here again, for the highest number of threads the MKL-LAPACK algorithm shows the best performance with 0.12 seconds. Our algorithm needs 0.39 seconds and for this setting we are 6 times faster than the PLASMA library, which needs 2.53 seconds and 10 times faster than auto-vectorization which needs 4.12 seconds.

7.6 Algorithm by Floyd and Warshall

For the experiments on the algorithm of Warshall (described in Section 6.4) we have generated a graph with 3 densely connected subsets of nodes (clusters). Two nodes within a cluster are connected with a probability of 1% and two nodes in separate clusters are not connected, so we have 3 connected components. In Figure 13 we compare our packed FUR-Hilbert approach to the canonical approach. For a number of 40 000 nodes our approach is 1.81 times faster than the canonical implementation, where our algorithms needs 6.18 seconds and the canonical implementation 11.19 seconds. The same speedup of 1.81 remains for the variation in the number of cores used, where our algorithm spends 5.33 seconds and the canonical implementation 9.62 seconds.

7.7 Energy Efficiency

Energy is an important resource in all kinds of computing systems and critical in case of cost and availability. We measure the energy consumption of our Intel Xeon E5 with the power meter "Power HiTester 3334", which supports power integration measurements. The power meter has an accuracy of $\pm 0.1\%$. We are using the serial port (RS-232) and our own C API to communicate with the power meter, which allows us to measure the algorithms power consumption without any initialization or finalization overhead.

Our algorithm is the most energy efficient one for the matrix multiplication, as illustrated in Figure 14 (left). For varying matrix sizes our algorithm slightly outperforms the other algorithms with one exception, where the problem size is 10 000. For other problem sizes the gain over MKL-BLAS is between 2% and 11% and over Peano ("TifaMMy") between 16% and a factor of 160%. We are up to 5 times more energy efficient than auto-vectorization, but we left it out in Figure 14 for clarity of presentation.

Figure 14 (middle) displays the energy efficiency of the cholesky decomposition. Here, MKL-LAPACK is clearly the most energy efficient algorithm. Nevertheless, our algorithm is between 2 and 5 times more energy efficient than PLASMA. We are approximately 7 times more energy efficient than auto-vectorization.

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

8 Related Work and Discussion

8.1 Cache-oblivious Algorithms

The notion of *cache-obliviousness* has been first introduced by Frigo et al. in 1999 [3]. A cache oblivious algorithm performs well on any type of multi-level memory hierarchy without knowing the structure and the parameters of the hierarchy, e.g. cache and memory size, transfer block size, and bandwidth.

Cache-oblivious algorithms [3] have attracted considerable attention as they are portable to almost all environments and architectures. Algorithms and data structures for basic tasks like sorting, searching, matrix multiplication as well as for specialized tasks like ray tracing [16] or homology search in bioinformatics [17] have been proposed. The two fundamental design patterns of cache-oblivious algorithms are localized memory access and divide-and-conquer. Space-filling curves (SFC) integrate both ideas. A SFC defines an 1D ordering of the points of an n-dimensional space such that each point is visited once. Probably the most widely used SFC is the Z-order due to its simple recursive scheme. The Hilbert and the Peano SFC provide better locality properties at the expense of more computational overhead. In order to theoretically formalize cache-obliviousness, Figo et al. introduced the assumption of an ideal cache which is characterized by ideal page replacement and full associativity. The authors prove that algorithms designed with this idealized setting in mind only degrade in performance by a constant factor in realistic settings such as LRU replacement strategy and set-associative caches. Most related to our work, Bader et al. proposed to use the Peano curve for matrix multiplication and LU-decomposition [11], [13]. Their algorithms process the input matrices in a block-wise and recursive fashion where the Peano curve guides the processing order

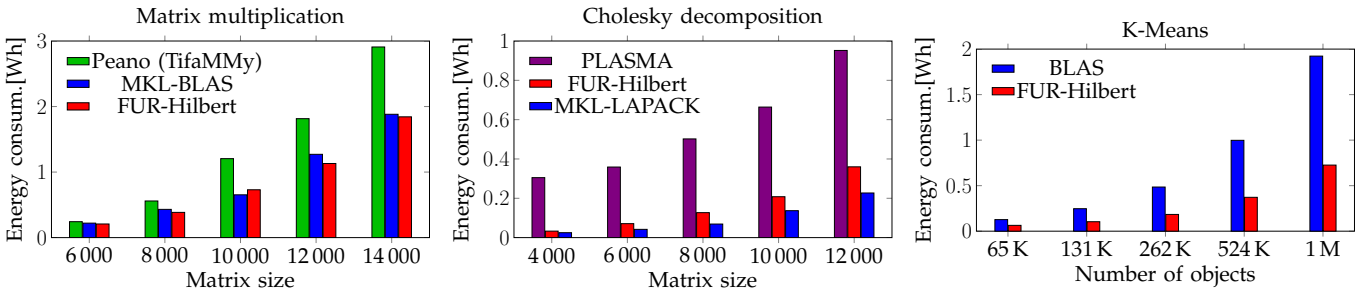

Fig. 14: Experiments on Energy Efficiency.

and thus the memory access pattern. We considerably improve memory locality and runtime by introducing the Fast Un-restricted Hilbert curve. Classical SFCs are restricted to traverse data of a specific size, e.g. multiples of powers of two or three. Bader et al. use the Peano curve with zero padding.

We introduce a Hilbert curve supporting arbitrary problem sizes which is of general interest also for other applications. We propose a highly efficient iterative algorithm to compute the Hilbert values on the fly. Most existing non-recursive approaches are based on lookup tables causing memory overhead, e.g., [5]. For determination of a single Hilbert value, most iterative techniques are linear in the resolution which corresponds to the number of iterations in the loop, thus causing substantial overhead. In contrast, our solution derecursivates the Lindenmayer-System [18] and introduces compact nano-programs fitting into registers for the traversal of small patches. Our approach processes a single pair of indices i, j in constant time, iterates through the loop in linear time and therefore causes negligible overhead. Recursive approaches, e.g., [6], [18] are associated with logarithmic worst-case time complexity for processing a single index pair and are not suitable to support comfortable loop programming, see also Section 1.

8.2 Optimized Techniques for Specific Tasks or Hardware

In [12], [19] Bader et al. present variants of their algorithms for matrix operations. As in [11], [13], the general algorithmic scheme is recursive partitioning according to the Peano curve. To tailor the algorithm to the properties of specific hardware, small matrix blocks are processed with optimized assembler code. For every microarchitecture, comprehensive adjustments are required to reach performance competitive with optimized libraries like BLAS [1], LAPACK [20], DAAL or MKL. The library BLAS (Basic Linear Algebra Subprograms) provides basic linear algebra operations together with programming interfaces to C and Fortran. Specific implementations for various infrastructures are available, e.g. ACML for for AMD Opteron processors or CUBLAS for NVIDIA GPUs. The Math Kernel Library (MKL) contains highly vectorized math processing routines for Intel processors. In contrast to our work, these implementations are very hardware-specific and mostly vendor-optimized. Moreover, they are designed to efficiently support specific linear algebra operations while we are aiming at supporting loop processing in general. Our experiments demonstrate that our cache-oblivious approach reaches a performance close to BLAS on the task of matrix multiplication and outperforms BLAS when applied to support K-means clustering.

As K-means probably is the most wide-spread clustering algorithm other highly optimized techniques for specific hardware have been proposed, e.g. for mobile devices [21], GPUs [22] or computing clusters [23]. Comparison to such specialized techniques is out of the scope of this work as K-means clustering only serves an exemplary host algorithm.

8.3 Energy Efficiency on Data Movement

The energy cost of data movement from memory to registers has been identified as one of the limiting factors for the development of efficient and sustainable exascale systems. The cost, in terms of energy, is two orders of magnitude higher than the cost of computing a double-precision register-to-register floating point operation [24].

In general a L3 cache miss is approximately three times more expensive than a L2 cache miss and a L2 cache miss is approximately three times more expensive than a L1 cache miss. In Figure 10 we have examined the cache footprint of the matrix multiplication, where our algorihm avoids expensive cache misses most effectively. For the matrix multiplication, we observe that our algorithm is slightly behind MKL-BLAS (cf. Figure 8) in cases of runtime, but slighthly ahead in cases of energy-efficiency (cf. Figure 14). We believe, that this is due to our efficient cache access pattern induced by our Hilbert curve. Figure 10 shows that our approach has a better cache hit rate for L2 and L3 caches.

9 Conclusion

We introduced Fast Unrestricted (FUR-) Hilbert, a new space-filling curve with the property that every arbitrary $n \times m$ rectangle can be filled by making only axis-parallel, single-step moves in a recursively bisected data space.

The original Hilbert-curve has the analogous property but is restricted to squares ($n = m$) where the side length n must be a power of two. In addition, our algorithm to generate the FUR-Hilbert curve is highly efficient because it is non-recursive and has a constant worst-case complexity per generated pair of coordinates in contrast to $O(\log n)$ for previous methods. These two advantages make it particularly attractive to replace pairs of nested loops in important host algorithms (matrix multiplication, K-Means clustering, Cholesky decomposition, Floyed-Warshall etc.) by our FUR-Hilbert curve which leads to cache-oblivious accesses of the corresponding objects. For future work, we plan to investigate if our cache-oblivious loops can accelerate algorithms in a distributed environment.

REFERENCES

[1] J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990.

[2] M. Alabduljalil, X. Tang, and T. Yang, "Cache-conscious performance optimization for similarity search," in *SIGIR Conference*, 2013, pp. 713–722.

[3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS Symposium*, 1999, pp. 285–298.

[4] C. Böhm, M. Perdacher, and C. Plant, "Cache-oblivious loops based on a novel space-filling curve," in *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, 2016, pp. 17–26.

[5] N. Chen, N. Wang, and B. Shi, "A new algorithm for encoding and decoding the hilbert order," *Softw., Pract. Exper.*, vol. 37, no. 8, pp. 897–908, 2007.

[6] G. Breinholt and C. Schierz, "Algorithm 781: Generating hilbert's space-filling curve by recursion," *ACM Trans. Math. Softw.*, vol. 24, no. 2, pp. 184–189, Jun. 1998.

[7] K. Chung, Y. Huang, and Y. Liu, "Efficient algorithms for coding hilbert curve of arbitrary-sized image and application to window query," *Inf. Sci.*, vol. 177, no. 10, pp. 2130–2151, 2007.

[8] M. Bläser, "Fast matrix multiplication," *Theory of Computing, Graduate Surveys*, vol. 5, pp. 1–60, 2013.

[9] C. Böhm, M. Perdacher, and C. Plant, "Multi-core k-means," in *Proceedings of the 2017 SIAM International Conference on Data Mining*, pp. 273–281.

[10] S. Warshall, "A theorem on boolean matrices," *J. ACM*, vol. 9, pp. 11–12, 1962.

[11] M. Bader and C. E. Mayer, "Cache oblivious matrix operations using peano curves," in *PARA Workshop*, 2006, pp. 521–530.

[12] A. Heinecke and C. Trinitis, "Cache-oblivious matrix algorithms in the age of multicores and many cores," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 9, pp. 2215–2234, 2010.

[13] M. Bader, "Exploiting the locality properties of peano curves for parallel matrix multiplication," in *Euro-Par Conference*, 2008, pp. 801–810.

[14] H. Jégou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 1, pp. 117–128, 2011.

[15] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[16] B. Moon, Y. Byun, T. Kim, P. Claudio, H. Kim, Y. Ban, S. W. Nam, and S. Yoon, "Cache-oblivious ray reordering," *ACM Trans. Graph.*, vol. 29, no. 3, 2010.

[17] M. Ferreira, N. Roma, and L. M. S. Russo, "Cache-oblivious parallel SIMD viterbi decoding for sequence search in HMMER," *BMC Bioinformatics*, vol. 15, p. 165, 2014.

[18] F. Fracchia, P. Prusinkiewicz, and A. Lindenmayer, *Synthesis of Space-filling Curves on the Square Grid*. Amsterdam, The Netherlands: Elsevier Sci. Pub. B. V., 1991.

[19] M. Bader, R. Franz, S. Günther, and A. Heinecke, "Hardware-oriented implementation of cache oblivious matrix operations based on space-filling curves," in *PPAM Conference*, 2007, pp. 628–638.

[20] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' Guide (Third Ed.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[21] F. An and H. J. Mattausch, "K-means clustering algorithm for multimedia applications with flexible HW/SW co-design," *Journal of Systems Architecture - Embedded Systems Design*, vol. 59, no. 3, pp. 155–164, 2013.

[22] J. Bhimani, M. Leeser, and N. Mi, "Accelerating k-means clustering with parallel implementations and GPU computing," in *HPEC Conference*, 2015, pp. 1–6.

[23] S. Shahrivari and S. Jalili, "Single-pass and linear-time k-means clustering based on mapreduce," *Inf. Syst.*, vol. 60, pp. 1–12, 2016.

[24] T. Wirtz, R. Ge, Z. Zong, and Z. Chen, "Power and energy characteristics of mapreduce data movements," in *International Green Computing Conference, IGCC 2013, Arlington, VA, USA, June 27-29, 2013, Proceedings*, 2013, pp. 1–7.

Christian Böhm is professor of informatics at Ludwig-Maximilians-Universität München, Germany. He received his Ph.D. in 1998 and his habilitation in 2001. His former affiliations include the Technische Universität München and UMIT, Hall in Tirol, Austria. His research focus is data mining, particularly index structures and clustering algorithms. He has received 4 research awards including the SIGMOD best paper award 1997 and the SIAM SDM best paper honorable mention award 2008.

Martin Perdacher Martin Perdacher received his B.Sc in Bioinformatics at the University of Applied Sciences Hagenberg in 2011 and his M.Sc in Scientific Computing at the University of Vienna in 2016. Martin is a Ph.D. student and his research focuses on high performance methods for exploratory data analysis.

Claudia Plant Claudia Plant received her Ph.D. degree in 2007 and currently is full professor for Data Mining at University of Vienna. Her research focuses on database-related data mining, especially clustering, parameter-free data mining, and integrative mining of heterogeneous data. She not only published in top-level database and data mining conferences like KDD and ICDM but also in application-related top-level journals. She received 3 awards including the ICDM best paper award 2014.