Journal of Parallel and Distributed Computing 138 (2020) 65–77

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Extending the limits for big data RSA cracking: Towards cache-oblivious TU decomposition

Fatima K. Abu Salem [a,*], Mira Al Arab [a], Laurence T. Yang [b]

[a] Computer Science Department, American University of Beirut, Beirut, Lebanon
[b] Department of Computer Science, St. Francis Xavier University, Antagonish, NS, Canada

ARTICLE INFO

Article history:
Received 17 January 2019
Received in revised form 25 June 2019
Accepted 24 December 2019
Available online 3 January 2020

Keywords:
Exact linear algebra
Cache-oblivious algorithms
Space-filling curves
Morton-hybrid order

ABSTRACT

Nowadays, Big Data security processes require mining large amounts of content that was traditionally not typically used for security analysis in the past. The RSA algorithm has become the de facto standard for encryption, especially for data sent over the internet. RSA takes its security from the hardness of the Integer Factorisation Problem. As the size of the modulus of an RSA key grows with the number of bytes to be encrypted, the corresponding linear system to be solved in the adversary integer factorisation algorithm also grows. In the age of big data this makes it compelling to redesign linear solvers over finite fields so that they exploit the memory hierarchy. To this end, we examine several matrix layouts based on space-filling curves that allow for a cache-oblivious adaptation of parallel TU decomposition for rectangular matrices over finite fields. The TU algorithm of Dumas and Roche (2002) requires index conversion routines for which the cost to encode and decode the chosen curve is significant. Using a detailed analysis of the number of bit operations required for the encoding and decoding procedures, and filtering the cost of lookup tables that represent the recursive decomposition of the Hilbert curve, we show that the Morton-hybrid order incurs the least cost for index conversion routines that are required throughout the matrix decomposition as compared to the Hilbert, Peano, or Morton orders. The motivation lies in that cache efficient parallel adaptations for which the natural sequential evaluation order demonstrates lower cache miss rate result in overall faster performance on parallel machines with private or shared caches and on GPU's.

1. Introduction

Nowadays, Big Data security processes require mining large amounts of content that was traditionally not typically used for security analysis in the past. The requirements of an intelligence-driven security system requires an ability to analyse vast streams of data from numerous sources to produce actionable information. To that end, a research uptake on how to leverage security mechanisms and models to handle large data is in required.

The RSA algorithm has become the de facto standard for encryption, especially for data sent over the internet. The RSA algorithm belongs to a class of algorithms that produce asymmetric keys. Those are easy to exchange securely, but have limited message size. In contrast, symmetric keys have an unlimited message size, but are difficult to exchange securely. The nature of the RSA algorithm is such that it is only able to encrypt a limited amount of plaintext. This is because the size of the modulus of an RSA key grows with the number of bytes to be encrypted. RSA takes its security from the hardness of the Integer Factorisation Problem. RSA crypto-algorithm is secure only if deducing the RSA key, i.e., the factorisation of the carefully chosen sufficiently large two prime numbers comprising the key, requires an enormous (super-polynomial) amount of time with respect to the size of the modulus number. As a result, much more research has been done to find ways to quickly factor such integer numbers, so a benchmark against the security of RSA can be established.

Leading integer factorisation algorithms based on sieving require solution of linear systems over the binary field (triangulating exact matrices) that grow with the size of the modulus of an RSA key. Despite that these systems begin as sparse, they evolve to be dense very early on in the process. For large linear systems over the binary field, there emerges a need to develop efficient exact direct solvers. A leading cause for inefficient mathematical software in the age of big data revolves around its ability to exploit the memory hierarchy. Big data platforms such as for Apache Spark require the development of cache-efficient algorithms and data structures as one of a handful of techniques required to bring its performance to push its performance closer to the limits of modern hardware [21]. Cache-efficient algorithms have been indispensable for targeting highly efficient mathematical kernels for numerical linear algebra, as we see in LAPACK and

* Corresponding author.
E-mail addresses: fa21@aub.edu.lb (F.K. Abu Salem), maa75@aub.edu.lb (M. Al Arab), ltyang@stfx.ca (L.T. Yang).

https://doi.org/10.1016/j.jpdc.2019.12.016
0743-7315/

BLAS. However, to the best of our knowledge, similar advances in the Symbolic community has been extremely rare. In this work, we address a major design question to render a cache-efficient adaptation of the TURBO algorithm of Dumas et al. [9] for exact LU decomposition. This algorithm recurses on rectangular and potentially singular matrices, which makes it possible to take advantage of cache effects. It improves on other expensive methods for handling singular matrices, which otherwise have to dynamically adjust the submatrices so that they become invertible. Particularly, TURBO significantly reduces the volume of communication on distributed architectures, and retains optimal work and linear span. TURBO can also compute the rank in an exact manner. As benchmarked against some of the most efficient exact elimination algorithms in the literature, TURBO incurs low synchronisation costs and reduces the communication cost featured by [14,15] by a factor of one third when used with only one level of recursion on 4 processors. In TURBO, local TU factorisations are performed until the sub-matrices reach a given threshold, and so one can take advantage of cache effects. A cache friendly adaptation of the serial version of TURBO bears impact on all possible forms of parallel or distributed deployment of the algorithm. For one, nested parallel algorithms with low depth and for which the natural sequential execution has low cache complexity will also attain good cache complexity on parallel machines with private or shared caches [5]. Locality of reference on distributed systems is also being advocated by the Databricks group initiated by founders of Apache Spark. In their own terms, when profiling Spark user applications on distributed clusters, a large fraction of the CPU time was spent waiting for data to be fetched from main memory. Locality of reference is also of concern on GPUs. Although one does not have full control over optimising locality of reference on such machines, and despite that GPUs rely on thread-level parallelism to hide long latencies associated with memory access, the memory hierarchy remains critical for many applications. Finally, applications that are cache aware are also deemed to be more energy aware, as remarked by the Green computing community. This preamble motivates our work on trying to improve the cache performance of the serial version of TURBO.

Our contributions can be summarised as follows:

1. We investigate prospects for a cache oblivious adaptation of the TURBO algorithm by mapping four different matrix layouts against each other: the Hilbert order [8,16], the Peano order [3,4], the Morton order [20,26], and the Morton-hybrid order [2]. Whilst matrices on which we want to perform matrix–matrix multiplication or LU decomposition without pivoting can be serialised no matter what layout is used, the recursive TU decomposition considered in this work consistently requires permutation steps that require one to traverse the matrix in a row-wise or column-wise manner, thus eliciting index conversion from the Cartesian scheme to the recursive scheme and vice versa.
2. Our analysis of the four schemes addresses the cost of bit operations and accessing table lookups when applicable. Our findings show the following:
   (a) The overhead for using the Peano layout will be compelling as index conversion invokes operations modulo 3.
   (b) Whilst the Hilbert layout has been promising for improving memory performance of matrix algorithms in general, and despite that the operations for encoding and decoding in this layout can be performed using bit shifts and bit masks, we will still require m iterations for a $2^m \times 2^m$ matrix for each single invocation of encoding or decoding.
   (c) In contrast, we find that the conversions for the Morton and the Morton-hybrid layouts incur a constant number of operations assuming the matrix is of dimensions at most $2^\alpha \times 2^\alpha$, where α is the machine word-size. For the typical value $\alpha = 64$, such matrix sizes are sufficiently large for many applications.
   (d) Furthermore, despite that the Morton order can be encoded and decoded faster than the Morton-hybrid order, the factor of improvement is constant: ten less operations. In return, the Morton-hybrid layout allows the recursion to stop when the blocks being divided are of some prescribed size equal to $T \times T$, thus decreasing the recursion overhead. These $T \times T$ blocks are stored in a row-major order, which allows for benefiting from compiler optimisations that have already been designed for this layout. The row-major ordering of the block at the base case also makes accessing the entries within the blocks at the base case of the inversion, multiplication, and decomposition steps of the algorithm faster and easier because no index conversion is required.
3. The present manuscript is an indispensable precursor for our work in [1], where we introduce the concepts of *alignment* of sub-matrices with respect to the cache lines and their *containment* within proper blocks under the Morton-hybrid layout, and describe the problems associated with the recursive subdivisions of TURBO under this scheme.

2. Background and related work

2.1. Space-filling curves

It is well established that traditional row-major or column-major layouts of matrices in compilers lead to extremely poor temporal and spatial locality of matrix algorithms. Instead, several matrix layouts based on space-filling curves have yielded cache-oblivious adaptations of matrix algorithms such as matrix–matrix multiplication [4,7] and matrix factorisation [3,11,27]. A space-filling curve is a linear traversal of a discrete, multi-dimensional space. For example, space-filling curves linearise access to elements in a 2D matrix structure. The first space-filling curves were proposed by Peano in [24]. Later on, Hilbert worked extensively on generating a curve that passes through all points of a square [13]. Later on, the Morton order was developed by Guy Macdonald Morton, defining a space-filling curve which is Z-shaped [22]. Space-filling curves possess desirable locality properties. For example, entries that are closed to each other in the two dimensional space have corresponding space-filling curve indices that are close to other too. Particularly, those alternative layouts to the row major order are recursive in nature and produce highly cache-oblivious adaptations that scale with the size of the underlying matrices. The cache-oblivious model does not require knowledge of, and hence tuning the algorithm according to, the cache parameters. Cache-oblivious programmes allow for resource usage not to be programmed explicitly, and for algorithms to be portable across varying architectures, as well as all levels of the memory hierarchy within one specific architecture. This aspect becomes more enticing considering that cache efficient parallel adaptations for which the natural sequential evaluation order demonstrates lower cache miss rate result in overall faster performance on parallel machines with private or shared caches and on GPU's. A number of works have explored the nexus between cache efficient layouts based on space-filling curves and a number of matrix algorithms. For example, the authors in [12] present the design and performance analysis of a cache-efficient algorithm for matrix multiplication on various

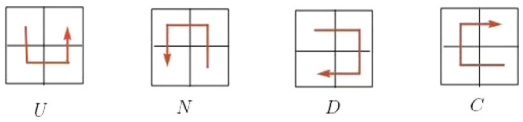

Fig. 1. Primitive Hilbert patterns.

shared-memory multicore systems based on Peano curves. The parallel implementation is based on OpenMP and tested on various shared memory platforms with up to 16 cores. In [19], the authors obtain a fast, double precision, general matrix–matrix multiplication algorithm for GPU architectures. Their design targets performance at the level of the L1 cache of a GPU, by considering different memory access patterns associated with the MM algorithm based on the Morton layout. All of these parallel implementations reveal superior performance in I/O performance as well as overall run-time performance, some of which have been benchmarked against state-of-the art libraries such as GotoBlas and Intel's MLK [12], thanks to the space-filling curves representations of the underlying matrices.

We now review the basics behind the widely used space-filling curve representations in the literature.

2.1.1. The Hilbert curve

A Hilbert curve traverses a $2n \times 2n$ array of points while never maintaining the same direction for more than three consecutive points, where n is termed the resolution or depth of the matrix. The standard, bit-processing algorithms for coding in the Hilbert curve are credited to Fosher, Liu and Schrack. Fisher's iterative algorithm uses two short look-up tables [10], and Liu and Schrack's faster algorithms avoid look-up tables [17,18].

The Hilbert ordering is recursively generated by dividing a $2^m \times 2^m$ matrix M into four quadrants following a certain pattern and recursively laying out the elements of these four quadrants [6]. We review the procedure to generate the Hilbert order of a $2^m \times 2^m$ matrix M, i.e. to map the entries of M to entries in the one-dimensional array representing M. The primitive patterns U, D, C and N are shown in Fig. 1.

In one variant of the Hilbert layout, the matrix M is assigned an initial pattern ρ_M from the four primitive patterns U, D, C, and N and its four quadrants are laid out in an order depending on the pattern ρ_M of M. To generate the Hilbert order of M, a recursive procedure is followed. We denote by Q^k the sub-matrix we are laying onto memory for each recursive level and denote by ρ_{Q^k} the pattern of the sub-matrix Q^k obtained from the set $\{U, D, C, N\}$. We start with $Q^0 = M$, assuming it has pattern $\rho_{Q^0} = U$. The generation proceeds as follows. At the kth step, each of the sub-matrices Q^k is refined into four quadrants $Q^{k+1} \in \{NW_{Q^k}, NE_{Q^k}, SW_{Q^k}, SE_{Q^k}\}$ of Q^k. These are then laid onto memory in the Hilbert order according to two rules: the *NextPattern* rule and the *HilbertOrder* rule. Let M_ρ denote any matrix M of pattern $\rho \in \{U, D, C, N\}$. Let $i \in \{0, 1, 2, 3\}$ denote the quadrants of M_ρ where 0, 1, 2, and 3 refer to the *NW*, *NE*, *SW*, *SE* quadrants respectively. The $NextPattern(M_\rho)$ rule for any sub-matrix M of pattern $\rho_M \in \{U, D, C, N\}$ identifies the pattern of each quadrant of M. The $HilbertOrder(M_\rho)$ rule identifies the order of precedence in which these quadrants are mapped onto memory. The *NextPattern* rule is:

$$NextPattern(M_\rho) = \begin{pmatrix} \rho_0 & \rho_1 \\ \rho_2 & \rho_3 \end{pmatrix}$$

where ρ_i is the pattern of the ith quadrant of M_ρ. The *HilbertOrder* rule is:

$$HilbertOrder(M_\rho) = \begin{pmatrix} v_0 & v_1 \\ v_2 & v_3 \end{pmatrix}$$

where v_i represents the order of precedence of quadrant i in the physical layout when generating the Hilbert order. The *NextPattern* rules for the four patterns as used in the generation of the Hilbert curve are given by the following:

$$NextPattern(M_U) = \begin{pmatrix} D & C \\ U & U \end{pmatrix} \qquad NextPattern(M_D) = \begin{pmatrix} U & D \\ N & D \end{pmatrix}$$

$$NextPattern(M_C) = \begin{pmatrix} C & U \\ C & N \end{pmatrix} \qquad NextPattern(M_N) = \begin{pmatrix} N & N \\ D & C \end{pmatrix}$$

The *HilbertOrder* rules for each pattern are given by the following:

$$HilbertOrder(M_U) = \begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix} \qquad HilbertOrder(M_D) = \begin{pmatrix} 0 & 1 \\ 3 & 2 \end{pmatrix}$$

$$HilbertOrder(M_C) = \begin{pmatrix} 2 & 3 \\ 1 & 0 \end{pmatrix} \qquad HilbertOrder(M_N) = \begin{pmatrix} 2 & 1 \\ 3 & 0 \end{pmatrix}$$

These rules can be deduced from the patterns shown in Fig. 1. In turn, Fig. 2 shows the steps of generating the Hilbert order for an 8×8 matrix M of initial pattern U. In Fig. 3, the entries of the matrix at the end of the generation of the Hilbert order show the indices of the elements of M within the physical one-dimensional array representing M in the Hilbert order. The matrix to be laid out in the Hilbert order is given a pattern and refined according to the refinement rules for each pattern. The refinement is done by recursively dividing the matrix into four quadrants and storing them according to the pattern refinement rules. Let M_ρ denote a Hilbert matrix of pattern $\rho \in \{U, D, C, N\}$. Each step of the refinement for M_ρ is done by identifying the patterns of the quadrants of M_ρ and the order in which these quadrants are accessed. This is done using the following structures for each matrix M_ρ laid out in some pattern ρ:

$$NextPattern(M_\rho) = \begin{pmatrix} \rho_0 & \rho_1 \\ \rho_2 & \rho_3 \end{pmatrix}, \quad HilbertOrder(M_\rho) = \begin{pmatrix} v_0 & v_1 \\ v_2 & v_3 \end{pmatrix},$$

where ρ_i denotes the pattern of the ith quadrant of M_ρ in the row-major layout, and v_i denotes the index in the Hilbert layout of the ith quadrant of M_ρ in the row-major layout. For example, the refinement rule for the U pattern is given by:

$$NextPattern(U) = \begin{pmatrix} D & C \\ U & U \end{pmatrix}, \quad HilbertOrder(U) = \begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix}.$$

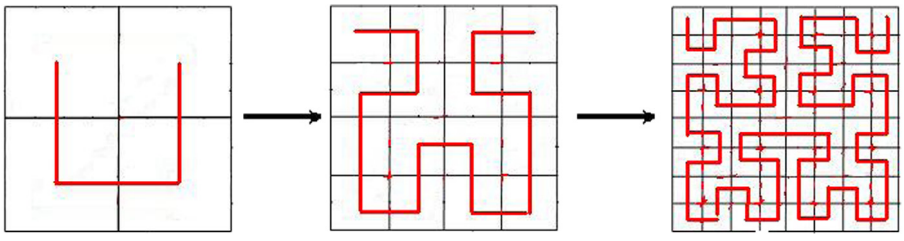

Fig. 2. Generation of Hilbert order.

0	3	4	5	58	59	60	63
1	2	7	6	57	56	61	62
14	13	8	9	54	55	50	49
15	12	11	10	53	52	51	48
16	17	30	31	32	33	46	47
19	18	29	28	35	34	45	44
20	23	24	27	36	39	40	43
21	22	25	26	37	38	41	42

Fig. 3. Matrix in the Hilbert order.

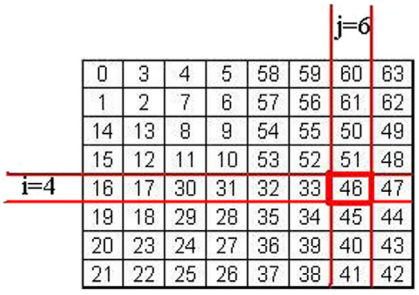

Fig. 4. Hilbert ordered matrix.

00	01	02
10	11	12
20	21	22

M

Fig. 5. Peano matrix division.

The refinement process is referred to as the generation of the Hilbert curve and is going to guide the encoding and decoding procedures. Fig. 4 shows an 8×8 matrix stored in the U-shaped Hilbert order on which the encoding and decoding procedures will be traced. In the rest of this section, Θ refers to the Hilbert order and M refers to a $2^m \times 2^m$ matrix in the Hilbert order.

2.2. The Peano curve

The Peano ordering of a matrix is based on the Peano space-filling curve. This ordering results from recursive construction

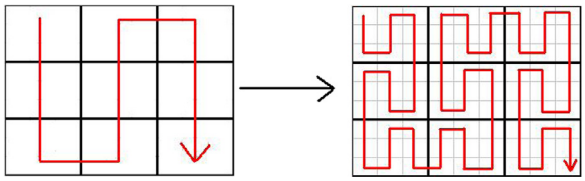

Fig. 7. Generation of Peano order.

0	5	6	47	48	53	54	59	60
1	4	7	46	49	52	55	58	61
2	3	8	45	50	51	56	57	62
15	14	9	44	39	38	69	68	63
16	13	10	43	40	37	70	67	64
17	12	11	42	41	36	71	66	65
18	23	24	29	30	35	72	77	78
19	22	25	28	31	34	73	76	79
20	21	26	27	33	33	74	75	80

Fig. 8. Matrix in the Peano order.

and assumes the matrix has dimension $3^m \times 3^m$ [4]. Each dimension of M is divided into 3 equal parts as shown in Fig. 5 resulting in nine sub-matrices which are named {00, 01, 02, 10, 11, 12, 20, 21, 22}. The Peano order for each of the resulting sub-matrices is recursively generated according to a certain precedence depending on the pattern of the higher level sub-matrix. Any Peano sub-matrix has a pattern from the set of primitive patterns $\{P, Q, R, S\}$ shown in Fig. 6. The Peano order assumes the initial matrix M has pattern P. We denote by Q^k the sub-matrix we are laying onto memory for each recursive level and denote by ρ_{Q^k} the pattern of the sub-matrix Q^k obtained from the set $\{P, Q, R, S\}$. We start with $Q^0 = M$ of pattern $\rho_{Q^0} = P$. At the kth step, each of the sub-matrices Q^k is refined into nine quadrants $Q^{k+1} \in \{00_{Q^k}, 01_{Q^k}, 02_{Q^k}, 10_{Q^k}, 11_{Q^k}, 12_{Q^k}, 20_{Q^k}, 21_{Q^k}, 22_{Q^k}\}$ of Q^k. These are then laid onto memory in the Peano order governed by two rules: the *NextPattern* rule and the *PeanoOrder* rule, similar to the rules governing the generation of the Hilbert order. Let M_ρ denote any matrix M of pattern $\rho \in \{P, Q, R, S\}$. Let $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ denote the sub-matrices of M_ρ where each i refers to the 00, 01, 02, 10, 11, 12, 20, 21, and 22 sub-matrices respectively. The *NextPattern*(M_ρ) rule for any sub-matrix M of pattern $\rho_M \in \{P, Q, R, S\}$ identifies the pattern of each quadrant of M. The *PeanoOrder*(M_ρ) rule identifies the order of precedence in which these quadrants are mapped onto memory. These rules can also be seen from Fig. 6: the precedence order of the sub-matrices is given by the direction of the arrow and the pattern is given by the letter inside the sub-matrix. Fig. 7 shows the steps for the generation of the Peano order for a 9×9 matrix M. In Fig. 8, the entries of the matrix at the end of the generation of the Peano order show the indices of the elements of M within the physical one-dimensional array representing M in the Peano order.

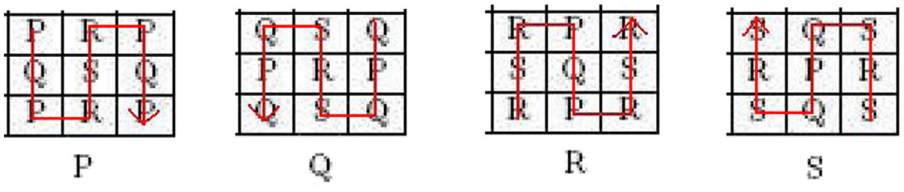

Fig. 6. Peano generation rules.

Fig. 9. Morton Z pattern.

2.3. The Morton curve

The Morton order is another alternative layout used for $2^m \times 2^m$ matrices. To generate the Morton-order of a matrix, the latter is divided into four quadrants, which are laid out onto memory in the order northwest, northeast, southwest, then southeast. The Morton order differs from the Hilbert order in that it does not alternate between patterns, but rather uses the same pattern to generate the Morton order for each of the sub-matrices. In this section, we present the Z-shaped Morton order, in which the pattern used looks like the letter Z as can be seen in Fig. 9. Fig. 10 shows the steps in the generation of the Z-shaped Morton order of an 8×8 matrix. For this example, there are three levels of decomposition, after which the entries of the matrix are laid out as shown in Fig. 11. In the remainder of this section, Θ refers to the Morton layout and we assume a given matrix M of dimensions $2^m \times 2^m$. Given a Cartesian index (i, j), recall that the length of the binary representation of i and j is at most m because $i, j \in \{0, 1, 2, \ldots, 2^m - 1\}$. These values can be represented by at most m bits. Write $i = (i_{m-1} \ldots i_4 i_3 i_2 i_1 i_0)_2$ and $j = (j_{m-1} \ldots j_4 j_3 j_2 j_1 j_0)_2$. According to [20], the corresponding Morton index z_Θ is

$$z_\Theta = (i_{m-1} j_{m-1} \ldots i_4 j_4 i_3 j_3 i_2 j_2 i_1 j_1 i_0 j_0)_2, \quad (1)$$

which represents an inter-leaving of the bits of i and j. The reverse process by which we extract the bits of i and j from the bits of z_Θ is referred to as de-leaving. In this section, we derive the costs for encoding and decoding the Morton order based on this inter-leaving of indices. For this, we revisit the notions of dilating and un-dilating integers from [20]. Dilating an integer $k = (k_{b-1} \ldots k_4 k_3 k_2 k_1 k_0)_2$ of b bits is the process of expanding k into an integer $k' = (0k_{b-1} \ldots 0k_4 0k_3 0k_2 0k_1 0k_0)_2$ of $2 \times b$ bits by inserting a zero bit between every two bits of k. Un-dilating is the reverse process that takes k' back to k.

2.4. The Morton-hybrid layout

The Morton-hybrid order is a variant of the Morton order used in [2] and [6] among others. In this variant of the Morton order, the recursive sub-dividing into blocks stops at a given truncation size T, resulting in a base case block of size $T \times T$, for which the row-major order is used. The value of the truncation size can vary.

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Fig. 11. Matrix in the Morton order.

To generate the Morton-hybrid order, the procedure for generation of the Morton order is followed, using the Z pattern as shown in Fig. 9, until the size of the sub-matrix being mapped onto memory is $T \times T$. To map the entries of this block in the one-dimensional array representing the matrix, a row-major order is assumed. Fig. 12 shows the steps in the generation of the Z-shaped Morton-hybrid order of an 16×16 matrix for truncation size $T = 4$. The resulting map of the entries of the matrix in the corresponding one-dimensional array in memory can be seen in Fig. 13.

The following presentation of the Morton-hybrid index for $T = 16$ was taken from [2], but we elaborate on it additionally in our Proposition 3.3 and the associated proof. For the remainder of this section, Θ denotes the Morton-hybrid layout for $T = 16$. We assume the input matrix M has dimension $2^m \times 2^m$. In Section 3.3, we saw that the Morton index is determined using the inter-leaving of the bits of i and j, for a given Cartesian index (i, j). It can be easily seen that e row major index was composed of the bits of j concatenated to the bits of i. The Morton-hybrid order is a combination of both these orders.

Consider the Morton-hybrid matrix shown in Fig. 16 morphing into row-major sub-blocks at the bast case for $T = 16$. Consider the row-major sub-block outlined in red in Fig. 16. Its row index is 1 and its column index is 2. The Morton index of a sub-block in the Morton-hybrid order is then given by the interleaving of the block row and block column indices as per the Morton indexing scheme. Now, each entry e in a Morton-hybrid matrix can be indexed with the help of information regarding (1) the $T \times T$ row-major sub-block in which it lies, (2) its row index within this row-major sub-block, and (3) its column index within this row-major sub-block. For each element e in a $T \times T$ sub-block, the possible values for the row and column indices of e within

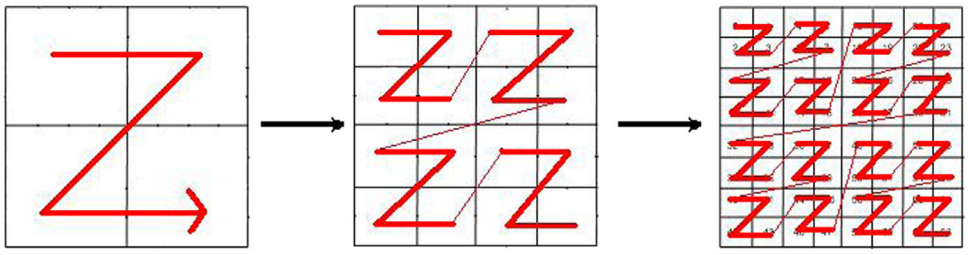

Fig. 10. Generation of Morton order.

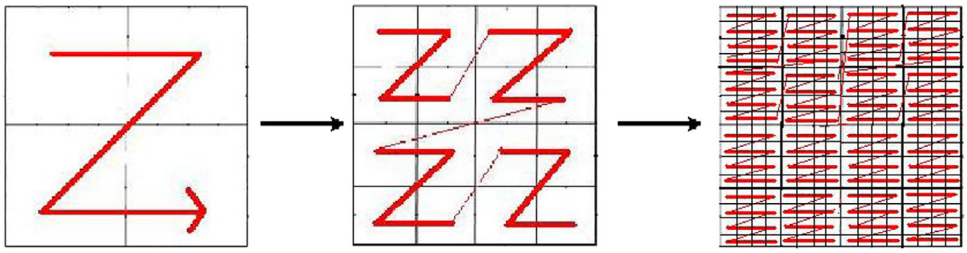

Fig. 12. Generation of Morton-hybrid order.

0	1	2	3	16	17	18	19	64	65	66	67	80	81	82	83
4	5	6	7	20	21	22	23	68	69	70	71	84	85	86	87
8	9	10	11	24	25	26	27	72	73	74	75	88	89	90	91
12	13	14	15	28	29	30	31	76	77	78	79	92	93	94	95
32	33	34	35	48	49	50	51	96	97	98	99	112	113	114	115
36	37	38	39	52	53	54	55	100	101	102	103	116	117	118	119
40	41	42	43	56	57	58	59	104	105	106	107	120	121	122	123
44	45	46	47	60	61	62	63	108	109	110	111	124	125	126	127
128	129	130	131	144	145	146	147	192	193	194	195	208	209	210	211
132	133	134	135	148	149	150	151	196	197	198	199	212	213	214	215
136	137	138	139	152	153	154	155	200	201	202	203	216	217	218	219
140	141	142	143	156	157	158	159	204	205	206	207	220	221	222	223
160	161	162	163	176	177	178	179	224	225	226	227	240	241	242	243
164	165	166	167	180	181	182	183	228	229	230	231	244	245	246	247
168	169	170	171	184	185	186	187	232	233	234	235	248	249	250	251
172	173	174	175	188	189	190	191	236	237	238	239	252	253	254	255

Fig. 13. Matrix in the Morton-hybrid order.

this block can be 0, 1, …, $T-1$. Thus, the offset values can be represented using β bits where $T = 2^{\beta}$.

2.5. The TU algorithm: A summary

Consider a $2m \times 2n$ matrix A with rank r over a field $\mathbb{F}$, where A may be singular. The TURBO algorithm triangulates the matrix A in a succession of recursive steps, relaxing the condition for generating a strictly lower triangular matrix. All the recursive steps are independent and thus TURBO is inherently parallel. It further outputs two matrices T and U, such that $A = T \cdot U$, where U is a $2m \times 2n$ upper triangular matrix, and T is $2m \times 2m$, with some "T" patterns. In all of the following, let $A^{(i,j)}_{(a,b)}$ denote the sub-matrix of A of dimensions $a \times b$ and starting at the entry of Cartesian index (i, j). Whenever the superscript (i, j) is omitted, a default value $(0, 0)$ is assumed. First, begin by decomposing the matrix A of size $2m \times 2n$ as follows:

$$A_{(2m,2n)} = \begin{pmatrix} NW_{(m,n)} & NE_{(m,n)} \\ SW_{(m,n)} & SE_{(m,n)} \end{pmatrix} \tag{2}$$

The TURBO algorithm now performs the following:

1. Recursive TU decomposition in each of SE, then SW, NE, and finally, NW
2. Virtual row and column permutations needed to re-order the blocks to yield a final, upper triangular matrix

For brevity, we only elaborate on the first step in the TURBO algorithm. It gives a flavour of the various matrix operations we will be addressing throughout the paper. The rest of the steps can be found in [9].

2.6. Step 1: Recursive TU in NW

This step performs a recursive call to the TU decomposition algorithm in the NW quadrant of A to get U_1 upper triangular, G_1, and L_1, lower triangular, such that

$$L_1 \cdot NW = \begin{pmatrix} U_1 & G_1 \\ 0 & 0 \end{pmatrix}.$$

Here, U_1 is $r \times r$ for some $r \geq 0$, G_1, and L_1 is $m \times m$. L_1 is then used to update NE by $B_{1_{(m,n)}} = L_1 \cdot NE$. One now aims to zero out the sub-matrix of SW that lies under U_1. This is done by calculating $N_{1_{(m,r)}} = -SW_{(m,r)} \cdot U_1^{-1}$, and then setting

$$\begin{pmatrix} 0_{(m,r)} & I_{1_{(m,n-r)}} \end{pmatrix} = SW + N_{1_{(m,r)}} \cdot \begin{pmatrix} U_{1_{(r,r)}} & G_{1_{(r,n-r)}} \cdot \end{pmatrix}$$

The submatrix SE has to be updated accordingly by $E_1 = SE + N_1 \cdot B_1$. At the end of this step, matrix A has been updated as follows:

$$A_1 = \begin{pmatrix} U_{1_{(r,r)}} & G_{1_{(r,n-r)}} & B_{1_{(r,n)}} \\ 0_{(m-r,r)} & 0_{(m-r,n-r)} & B_{1_{(m-r,n)}} \\ 0_{(m,r)} & I_{1_{(m,n-r)}} & E_{1_{(m,n)}} \end{pmatrix}.$$

The resulting matrix can be seen in Fig. 14(a) taken from [9].

3. Cross-analysis of index conversion overhead

The encoding and decoding procedures are those processes needed to revert between a space-filling curve index z and the corresponding Cartesian index (x, y). To illustrate, let Θ denote a subscript associated with one of the four layouts named above. Given a Cartesian index (i, j), encoding it in the order Θ corresponds to calculating its index z_Θ in the resulting matrix layout under Θ. Given an index z_Θ of a matrix entry under order Θ, decoding z_Θ corresponds to calculating the Cartesian index (i, j). Table 1 is a list of terms used in the remainder of this manuscript:

Additionally, we shall refer to the index z_Θ of an entry in a given matrix laid out in order Θ to denote its offset within the linear array representing the matrix in the layout Θ. Also, we shall refer to the index of a sub-block of size $2^k \times 2^k$ in a given matrix laid out in order Θ as the offset of this sub-block within

Table 1
Terms and Definitions.

Term	Definition
&	Bitwise AND operator
|	Bitwise OR operator
$\ll$	Bitwise left shift operator, where $i \ll k$ is equivalent to multiplying i by 2^k
$\gg$	Bitwise right shift operator, where $i \gg k$ is equivalent to dividing i by 2^k
Masking a value v	Applying an AND operation on v and a mask to extract part of v
Row index of a matrix entry e	Its row offset i within the matrix, derived from the Cartesian index (i, j) of e. The column index of the entry is its column offset, given by j
Masks	Bit values represented in hexadecimal format
Row index of a sub-block of matrix M	Its row offset within M and its column index is its column offset within M

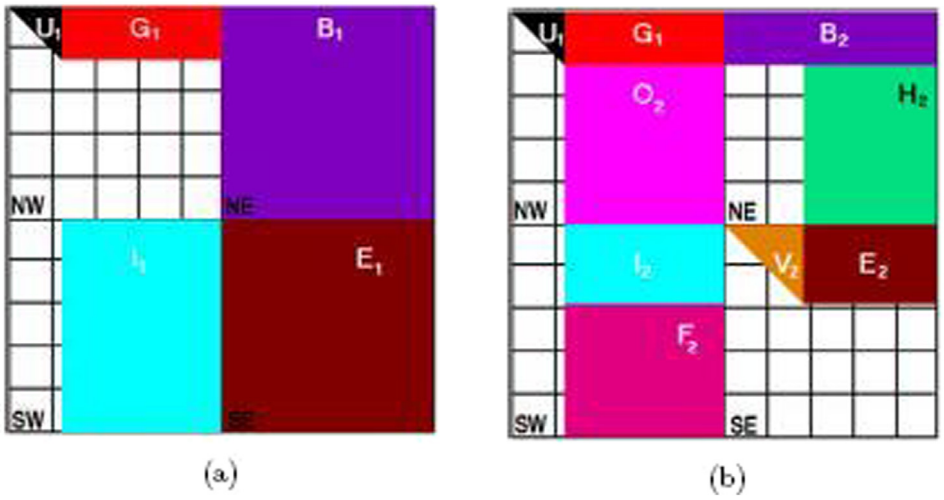

Fig. 14. Matrix after Step 1 (a) and Step 2 (b).

the linear array of objects representing the matrix in the layout Θ, where each object is a $2^k \times 2^k$ sub-block.

Finally, the masks we use are the following:

- $0x00FF00FF = (00000000111111110000000011111111)_2$
- $0x0F0F0F0F = (00001111000011110000111100001111)_2$
- $0x33333333 = (00110011001100110011001100110011)_2$
- $0x55555555 = (01010101010101010101010101010101)_2$
- $0x0000FFFF = (00000000000000001111111111111111)_2$
- $0xAAAAAAAA = (10101010101010101010101010101010)_2$

3.1. Encoding/decoding in the Hilbert-based layout

3.1.1. Encoding in the Hilbert order

Given an entry e with Cartesian index (i, j) in the $2^m \times 2^m$ matrix M, we denote by Q_e^{k+1} the quadrant of Q_e^k of dimensions $2^{(m-(k+1))} \times 2^{(m-(k+1))}$ and in which the entry e lies in refinement step k, for $k = 0, 1, ..m-1$. We start with $Q_e^0 = M$. To use the refinement rules, we translate them into two lookup tables: Table $\mathcal{T}_\mathcal{P}$ and Table $\mathcal{T}_\mathcal{V}$. We index the lookup tables using ρ and v: ρ is the pattern of the matrix we are refining and v is the index, in the row-major order, of the quadrant Q_e^{k+1} within Q_e^k. Table $\mathcal{T}_\mathcal{P}$ is a table mapping a pattern ρ and an index v to the next pattern, i.e. the pattern of the block Q_e^{k+1}, and is given by Table 2. The other look-up table, $\mathcal{T}_\mathcal{V}$, maps a pattern ρ and an index v to two bits of z_Θ and is given by Table 3. These two bits are the Hilbert index of Q_e^{k+1} within Q_e^k. To determine these tables, consider matrix M_ρ of pattern $\rho \in \{U, D, C, N\}$. Let $v \in \{0, 1, 2, 3\}$ refer to the row-major index of any quadrant of M, designating NW, NE, SW, and SE respectively. The entry $\mathcal{T}_\mathcal{P}(\rho, v) = \rho'$ is determined as follows: ρ' is the pattern of the vth quadrant of M_ρ in the row-major order. Table $\mathcal{T}_\mathcal{V}$ presents a mapping between the row-major layout of the quadrants of a matrix of pattern ρ and the Hilbert layout of these quadrants. The entry $\mathcal{T}_\mathcal{V}(\rho, v) = v'$ is the Hilbert index of the vth quadrant of M_ρ in the row-major order. For example, recall that a matrix of the U Hilbert pattern is refined as follows:

$$NextPattern(M_U) = \begin{pmatrix} D & C \\ U & U \end{pmatrix}, \quad HilbertOrder(M_U) = \begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix}.$$

Hence the entry $\mathcal{T}_\mathcal{P}(U, 1)$ is C and $\mathcal{T}_\mathcal{V}(U, 1) = 3 = (11)_2$.

Recall that Q_e^k denotes the $2^{(m-k)} \times 2^{(m-k)}$ quadrant of Q_e^{k-1} in which the element e of Cartesian index (i, j) lies at each refinement step. Table $\mathcal{T}_\mathcal{P}$ guides the generation of z_Θ by identifying the pattern of Q_e^{k+1}. Table $\mathcal{T}_\mathcal{V}$ identifies two bits to be appended at each iteration to a binary index z_k. These two bits represent the index, in the Hilbert layout, of Q_e^{k+1} within Q_e^k. We justify that this index is identified using two bits as follows. Recall that Q_e^{k+1} is one of the quadrants of Q_e^k. There are only four quadrants in a matrix. Hence, the possible values for the index of any of these quadrants – regardless of the layout – are 0, 1, 2, and 3, which can be represented using at most two bits. Hence, the length of the binary representation of this index is at most two.

Table 2
Encoding pattern look-up table $\mathcal{T}_\mathcal{P}$.

	0	1	2	3
U	*D*	*C*	*U*	*U*
C	*C*	*U*	*C*	*N*
D	*U*	*D*	*N*	*D*
N	*N*	*N*	*D*	*C*

Table 3
Encoding bits look-up table $\mathcal{T}_\mathcal{V}$.

	0	1	2	3
U	00	11	01	10
C	10	11	01	00
D	00	01	11	10
N	10	01	11	00

Now that the lookup tables are ready, we describe the iterative encoding procedure for a given $2^m \times 2^m$ matrix and a Cartesian index (i, j). Each iteration represents a refinement step within the generation of the Hilbert curve and, after m iterations, the sequence $\{z_{k+1}\}_{k=0,1,...,m-1}$, converges to z_Θ. In each iteration k, the pattern ρ_k - whether U, C, D, or N - of the quadrant Q_e^{k+1} of Q_e^k must be determined. Recall that Q_e^k is the $2^{(m-k)} \times 2^{(m-k)}$ sub-block of M containing element e of Cartesian index (i, j). The initial Q_e^0 is M and $\rho_0 = U$ because the Hilbert pattern we assume for the initial matrix is U. The corresponding index in the Hilbert order is progressively calculated by finding some partial index z_k in each iteration. The algorithm begins with $z_0 = 0$ and ends with the value for $z_m = z_\Theta$. Write $i = (i_{m-1}...i_2 i_1 i_0)_2$ and $j = (j_{m-1}...j_2 j_1 j_0)_2$. Recall that there are m bits in each of i and j, because at most m bits are needed to represent a value between 0 and 2^{m-1} in the binary system. In each iteration k of the encoding algorithm:

- we generate an auxiliary index $v_k = (i_{(m-1-k)} j_{(m-1-k)})_2$, where $v_k \in \{0, 1, 2, 3\}$, the two-bit, row-major index of Q_e^{k+1} within Q_e^k.

- we use v_k and ρ_k to index the lookup table $\mathcal{T}_{\mathcal{P}}$ and find the next pattern of Q_e^{k+1} $\rho_{k+1} = \mathcal{T}_{\mathcal{P}}(\rho_k, v_k)$.
- we use v_k and ρ_k to index the lookup table $\mathcal{T}_{\mathcal{V}}$ and find

  $z_{k+1} = (z_k \ll 2)|\mathcal{T}_{\mathcal{V}}(\rho_k, v_k)$.

  This appends the two bits of the Hilbert index of Q_e^{k+1} within Q_e^k to the partial index z_k to get z_{k+1}.

As two bits are appended to z_k at each iteration, and there are m iterations, then z_Θ is formed of $2m$ bits. To prove correctness, we propose the following:

Lemma 3.1. *The auxiliary variable v_k is the row-major index of Q_e^{k+1} in Q_e^k.*

Proof. Consider the entry e of Cartesian index (i, j). Write $i = (i_{m-1}...i_2 i_1 i_0)_2$ and $j = (j_{m-1}...j_2 j_1 j_0)_2$. To prove that v_k is the row-major index of Q_e^{k+1} within Q_e^k, we will show that the bits $i_{(m-1-k)}$ and $j_{(m-1-k)}$ are the row and column indices of the Q_e^{k+1} within Q_e^k respectively, from which it follows directly that v_k, the concatenation of $j_{(m-1-k)}$ to $i_{(m-i-k)}$, is the row-major index of Q_e^{k+1} within Q_e^k. We proceed by induction on k.

Base Case: For $k = 0$, recall that Q_e^0 is the $2^m \times 2^m$ matrix M. Let i_B denote the row index of Q_e^1 within Q_e^0. We need to show that $i_B = i_{(m-1)}$. Let i_r denote the row index of e within Q_e^1. The row index i of e is given by $i = i_B \times 2^{m-1} + i_r$ since Q_e^1 is of size $2^{m-1} \times 2^{m-1}$. We also have:

$$(i = i_B \times 2^{m-1} + i_r) \leftrightarrow (i = (i_B \ll (m-1))|i_r).$$

Hence, i_B is the $(m-1)^{st}$ bit of i given by $i_{(m-1)}$, which establishes the base case.

Inductive Step: We want to show that $i_{(m-(k+1))}$ is the row index of Q_e^{k+1} within Q_e^k. Let i_B denote the row index of Q_e^{k+1} within Q_e^k. We need to show that $i_B = i_{(m-(k+1))}$. Let i_r and i'_r denote the row index of e within Q_e^k and Q_e^{k+1} respectively. As Q_e^{k+1} is a $2^{m-(k+1)} \times 2^{m-(k+1)}$ quadrant within Q_e^k, we have

$$(i_r = i_B \times 2^{m-(k+1)} + i'_r) \leftrightarrow (i_r = (i_B \ll (m-(k+1)))|i'_r). \quad (3)$$

By induction, we know that the bit $i_{(m-1-k)}$ is the row index of Q_e^k within Q_e^{k-1}, so that

$$i = (i_{m-1} i_{m-2}...i_{(m-k)} \ll (m-k))|i_r. \quad (4)$$

Replacing i_r as in Eq. (3) in Eq. (4) above, we obtain

$$i = (i_{m-1} i_{m-2}...i_{(m-k)} \ll (m-k))|(i_B \ll (m-(k+1)))|i'_r.$$

Hence the $i_{(m-(k+1))} =_B$. This establishes the inductive step and so the proof.

Proposition 3.2. *The partial index z_k is the Hilbert index of the sub-block Q_e^k of dimensions $2^{(m-k)} \times 2^{(m-k)}$ within M, for $k = 0, 1, \ldots, m$.*

Proof. We proceed by induction on k.

Base Case: For $k = 0$, $Q_e^k \equiv M$ and has index $z_k = z_0 = 0$ within the Hilbert layout of M.

Induction Step: We assume that z_k is the Hilbert index of the $2^{(m-k)} \times 2^{(m-k)}$ sub-block Q_e^k within M. We refine more and obtain the $2^{(m-(k+1))} \times 2^{(m-(k+1))}$ quadrant Q_e^{k+1} within Q_e^k. From Lemma 3.1, we know that v_k is the row-major index of Q_e^{k+1} within Q_e^k. The quadrant Q_e^{k+1} has Hilbert index $v'_k = \mathcal{T}_{\mathcal{V}}(\rho_k, v_k)$ within Q_e^k from the definition of table $\mathcal{T}_{\mathcal{V}}$. that $z_{k+1} = (z_k \ll 2)|v'_k$ is the Hilbert index of Q_e^{k+1} within M. By induction, z_k represents the Hilbert index of Q_e^k within M. Upon another refinement step, each of the $2^{(m-k)} \times 2^{(m-k)}$ sub-blocks of M is in turn divided into four quadrants of size $2^{(m-(k+1))} \times 2^{(m-(k+1))}$ each. The indices of the four quadrants of Q_e^k are obtained by $z_k \times 4$, $z_k \times 4+1$, $z_k \times 4+2$, and $z_k \times 4+3$ depending on their index v'_k in the Hilbert layout of Q_e^k. These indices can be re-written as $(z_k \ll 2)|00$, $(z_k \ll 2)|01$, $(z_k \ll 2)|10$, and $(z_k \ll 2)|11$. The sub-block Q_e^{k+1} is a quadrant of Q_e^k and thus Q_e^{k+1} takes on one of these indices depending on its Hilbert index v'_k within Q_e^k. Thus Q_e^{k+1} has index

$$z_{k+1} = (z_k \ll 2)|v'_k$$

within M. This concludes the proof.

As an immediate result of the above, we get that the index z_m is z_Θ.

Table 4
Decoding pattern look-up table $\mathcal{T}'_{\mathcal{P}}$.

	0	1	2	3
U	*D*	*U*	*U*	*C*
C	*N*	*C*	*C*	*U*
D	*U*	*D*	*D*	*N*
N	*C*	*N*	*N*	*D*

Table 5
Decoding bits look-up table $\mathcal{T}'_{\mathcal{V}}$.

	0	1	2	3
U	00	10	11	01
C	11	10	00	01
D	00	01	10	11
N	11	01	00	10

3.1.2. Decoding in the Hilbert order

In the decoding procedure, we are given the Hilbert index z_Θ of an entry e and we wish to find (i, j), the Cartesian index of e. Recall that Q_e^k denotes the quadrant of Q_e^{k-1} in which e lies, starting with $Q_e^0 = M$. To find (i, j), we need two different lookup tables: $\mathcal{T}'_{\mathcal{P}}$ (Table 4) is used to identify the pattern of Q_e^k. Table $\mathcal{T}'_{\mathcal{V}}$ (Table 5) is used to find a two-bit value v'_k representing the row-major index of Q_e^k within Q_e^{k-1}. To determine these tables, consider matrix M_ρ of pattern $\rho \in \{U, D, C, N\}$. Recall that the *NW*, *NE*, *SW*, and *SE* quadrants are the 0th, 1st, 2nd, 3rd quadrants in the row-major order. Entries in Table $\mathcal{T}'_{\mathcal{P}}$ represent patterns and are given by $\mathcal{T}'_{\mathcal{P}}(\rho, v) = \rho'$. The pattern ρ' refers to the pattern of the vth quadrant of M_ρ in the Hilbert order. Table $\mathcal{T}'_{\mathcal{V}}$ presents a mapping between the Hilbert layout of the quadrants of M_ρ and the row-major layout of these quadrants. The entry $\mathcal{T}'_{\mathcal{V}}(\rho, v) = v'$ indicates that the vth quadrant of M_ρ in the Hilbert order is the v'^{th} quadrant of M_ρ in the row-major order. Note that $\mathcal{T}'_{\mathcal{V}}$ is the inverse mapping of the table $\mathcal{T}_{\mathcal{V}}$ used for encoding: i.e.

$$\mathcal{T}'_{\mathcal{V}}(\rho, v) = v' \leftrightarrow \mathcal{T}_{\mathcal{V}}(\rho, v') = v.$$

For example, recall that a matrix M_U in a U Hilbert pattern is refined as follows:

$$NextPattern(M_U) = \begin{pmatrix} D & C \\ U & U \end{pmatrix}, \quad HilbertOrder(M_U) = \begin{pmatrix} 0 & 3 \\ 1 & 2 \end{pmatrix}.$$

Hence, we have $\mathcal{T}'_{\mathcal{P}}(U, 3) = C$ because the 3rd quadrant of M_U in the Hilbert order has pattern C. Also, $\mathcal{T}'_{\mathcal{V}}(U, 3) = 1 = (01)_2$ because the 3rd quadrant of M_U in the Hilbert order is the 1st - i.e. the *NE* quadrant of M_U in the row-major order. Tables $\mathcal{T}'_{\mathcal{P}}$ and $\mathcal{T}'_{\mathcal{V}}$ guide the generation of i and j by identifying the pattern of Q_e^k and the row-major index of Q_e^k within M respectively.

Now that the lookup tables have been described, we present the decoding procedure. Recall from the encoding procedure that z_Θ is formed of $2m$ bits. Write $z_\Theta = (z_{2m-1}...z_2 z_1 z_0)_2$. The decoding reverses the encoding procedure, in which at each iteration

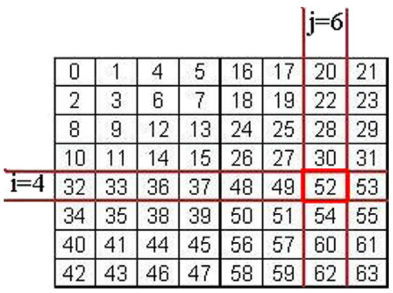

Fig. 15. Morton ordered matrix.

one bit of each of i and j is used to generate two bits of z. Recall, from encoding, that, in each iteration k, these two bits of z_Θ make up the Hilbert index of Q_e^k within Q_e^{k-1} as follows:

$v_k = (z_{(2m-1-2k)}z_{(2m-1-(2k+1))})_2$.

Also, we identify the pattern ρ_k of Q_e^k. We start with $\rho_0 = U$. In each iteration k, ρ_k and v_k are used to index the two lookup tables $\mathcal{T}'_{\mathcal{P}}$ and $\mathcal{T}'_{\mathcal{V}}$. As mentioned earlier, table $\mathcal{T}'_{\mathcal{P}}$ is used to find the pattern of the quadrant of the next iteration Q_e^{k+1}, given by $\rho_{k+1} = \mathcal{T}'_{\mathcal{P}}(\rho_k, v_k)$. Table $\mathcal{T}'_{\mathcal{V}}$ is used to find a two-bit value $v'_k = (b_i b_j)_2$ which represents the row-major index of Q_e^{k+1} within Q_e^k. Recall that this is a two-bit value because it represents the index of a quadrant. The values of i and j are found progressively: start with $i_0 = 0$ and $j_0 = 0$. In each iteration append $b_i \in \{0, 1\}$ and $b_j \in \{0, 1\}$ to i_k and j_k to get i_{k+1} and j_{k+1} respectively. In bit operations, this corresponds to $i_{k+1} = (i_k \ll 1)|(v'_k \gg 1)$, which appends bit b_i to i_k to get i_{k+1}, and $j_{k+1} = (j_k \ll 1)|(v'_k \& 1)$, which appends bit b_j to j_k to get j_{k+1}. This reverses the process of encoding. We iterate m times, after which we have $i_m = i$ and $j_m = j$.

3.2. Encoding/decoding in the Peano layout

The Peano-based ordering of matrices has the property that computations within matrix–matrix multiplication can be re-ordered so as to ensure no jumps in the address space [4]. The Peano layout is similar to the Hilbert layout in that it is based on more than one pattern, and so the encoding and decoding procedures for the Peano layout are analogous to those for the Hilbert layout. Specifically, the indices are encoded (or decoded) progressively through iterations. In each iteration, a pattern identifier is used along with a set of lookup tables to identify auxiliary variables for the next iteration. The differences between the procedure for the Peano layout and those for the Hilbert layout are:

- The operations used in each iteration are operations in base 3 - division by 3 and modulo 3.
- The resulting indices are in base 3 and need to be converted to base 2.

3.3. Encoding/decoding in the Morton layout

Before addressing the computational overhead for index conversion, we review two algorithms - *dilate* and *un-dilate* - that perform the encoding and decoding between Morton order indices and Cartesian indices. Fig. 15 shows an 8×8 Morton ordered matrix we will be using as example.

3.3.1. Encoding in the Morton order

All the present discussion follows from [20,26]. For encoding the Morton order, z_Θ must be found given (i, j). As shown in

Algorithm 1 (unsigned int) *dilate*(unsigned short t) [26]

```
unsigned int r = t
r = (r | (r << 8)) & 0x00FF00FF
r = (r | (r << 4)) & 0x0F0F0F0F
r = (r | (r << 2)) & 0x33333333
r = (r | (r << 1)) & 0x55555555
return r
```

Eq. (1), the inter-leaving of the bits of i and j must be performed. This can be done by finding the dilated forms of i and j, denoted by i' and j' respectively. The algorithm for dilation is given by Alg. 1 from [26]. According to the definition of dilation above, dilating i and j gives $i' = (0i_{m-1}0i_{m-2}...0i_30i_20i_10i_0)_2$ and $j' = (0j_{m-1}0j_{m-2}...0j_30j_20j_10j_0)_2$. Then z_Θ is given by

$$z_\Theta = (i' \ll 1)|j'. \tag{5}$$

To verify Eq. (5) from [20], perform

$i' \ll 1 = (i_{m-1}0i_{m-2}0...i_30i_20i_10i_00)_2$

followed by

$(i' \ll 1)|j' = (i_{m-1}j_{m-1}i_{m-2}j_{m-2}...i_3j_3i_2j_2i_1j_1i_0j_0)_2$.

3.3.2. Decoding in the Morton order

To decode the Morton order, we need to extract i and j from z_Θ. We write z_Θ as in Eq. (1). Then, the bits of z_Θ must be de-leaved to separate the bits of i, denoted i_{z_Θ}, from the bits of j, denoted j_{z_Θ}. To do this, we mask z_Θ with 0x*AAAAAAAA* to get $i_{z_\Theta} = (i_{m-1}0i_{m-2}0...i_30i_20i_10i_00)_2$ and mask z_Θ with 0x55555555 to get $j_{z_\Theta} = (0j_{m-1}0j_{m-2}...0j_30j_20j_10j_0)_2$. Now we shift i_{z_Θ} one position to the right to get

$i'_{z_\Theta} = i_{z_\Theta} \gg 1 = (0i_{m-1}0i_{m-2}...0i_30i_20i_10i_0)_2$.

To calculate i, we un-dilate i'_{z_Θ} using the un-dilation algorithm given in Alg. 2. To calculate j, we un-dilate j_{z_Θ} using Alg. 2.

Algorithm 2 unsigned short *un-dilate*(unsigned int t) taken from [20,26]

```
unsigned int r = t
t = (t | (t >> 1)) & 0x33333333
t = (t | (t >> 2)) & 0x0F0F0F0F
t = (t | (t >> 4)) & 0x00FF00FF
t = (t | (t >> 8)) & 0x0000FFFF
return (unsigned short)t
```

3.4. Encoding/decoding in the Morton-hybrid layout

Let (i, j) denote the Cartesian index of e. To be able to explore the coding overheads associated with the Morton-hybrid order, we require with the following results:

Proposition 3.3. *The β lower order bits of i represent the row index of the element within the $T \times T$ row-major sub-block and the $(m - \beta)$ higher order bits of i represent the row index of this $T \times T$ sub-block.*

Proof. Let M be a $2^m \times 2^m$ matrix laid out in the Morton-hybrid order with truncation size $T = 2^\beta$. Let e be any entry in the $T \times T$ row-major sub-block S_M of M, and let (i, j) denote the Cartesian index of e. First we show that the bit representations of the row index of the sub-block S_M and the row index of the element e within S_M require $(m - \beta)$ and β bits respectively.

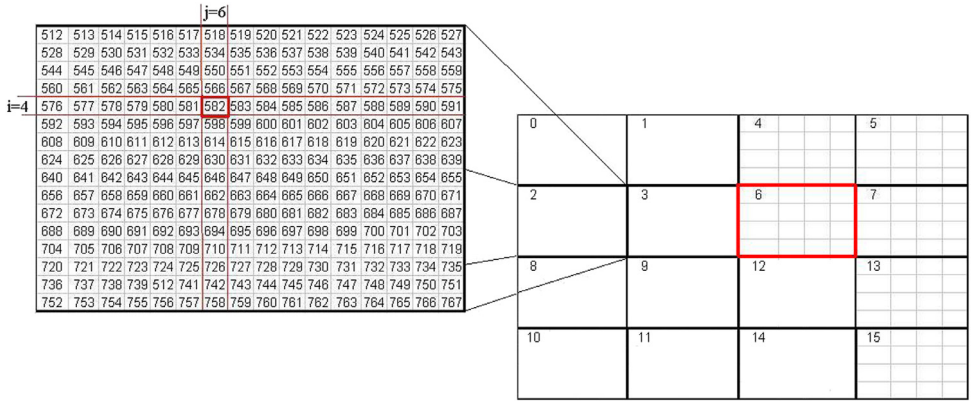

Fig. 16. Morton-Hybrid ordered matrix with $T = 16$. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The matrix M has dimensions $2^m \times 2^m$ and the sub-blocks at the base case have dimensions $T \times T$, with $T = 2^\beta$. Thus there are exactly $2^{(m-\beta)}$ rows of $T \times T$ blocks and the row index i_M of a sub-block can be 0, 1, …, $2^{(m-\beta)} - 1$. So these row indices require $(m - \beta)$ bits to be represented. The $T \times T$ sub-block S_M of M is in the row-major layout and contains T rows of entries. Let r denote the row of S_M in which entry e lies. The row index i_r of e within S_M, which is the same as the row index of any entry within r, can be 0, 1, …, $T - 1$ and thus its bit representation requires β bits.

Now we identify where to get these bits from. The row index i of the entry e within M is given by $i = i_M \times T + i_r$, which is equivalent to $i = (i_M \ll \beta)|i_r$. As such, i can be represented by m bits: the higher order $(m - \beta)$ bits are the bits of i_M, making up the row index of S_M within M, and the β lower order bits are the bits of i_r, making up the row index of e within S_M. This concludes the proof.

To illustrate the proposition, take for example the element (20, 6) from Fig. 16. In this example, $n = 2^m = 64$, i.e. $m = 6$, and $T = 2^\beta = 16$, so $\beta = 4$. Consider $i = 20 = (010100)_2$. The $\beta = 4$ lower order bits are $(0100)_2 = 4$, the element's row index within the row-major block, i_r. The $m - \beta = 2$ higher order bits $(01)_2 = 2$ identify the row index of the row-major sub-block in which this element lies. The same applies to $j = 6 = (000110)_2$. The $\beta = 4$ lower order bits $(0110)_2 = 6$ are the column index of the element within the row-major block. The $m - \beta = 2$ higher order bits, $(00)_2$, correspond to the column index of the block.

Now consider a matrix M laid out in the Morton-hybrid order and an entry e in M. Let z_Θ denote the Morton-hybrid index of e and (i, j) denote its Cartesian index. Let S_M denote the $T \times T$ row-major sub-block in which e lies. Recall the interleaving of the two coordinates i and j presented in Section 3.3 leading up to the Morton index of e. The $T \times T$ sub-blocks of M are laid out in the Morton order, so we use this procedure to inter-leave the $(m - \beta)$ higher order bits of i and j to get the Morton index z_M of S_M. Then, we find the row-major index of e within S_M: as S_M is stored in row-major order, the index within S_M of e is in the context of a row-major ordering scheme. To get the row-major index of e within S_M, denoted by z_r, we concatenate j_r to i_r, where i_r and j_r are the row and column indices of element e within S_M respectively. Because the smallest sub-blocks of M are of size $T \times T$, the Morton-hybrid index of the element e of Cartesian index (i, j) is then given by $(z_M \times T^2) + z_r$, or $(z_M \ll 2 \cdot \beta)|z_r$. Hence, it can be formed by concatenating the row-major offset of e within S_M to the Morton index of S_M within M. Recall, from Proposition 3.3, that the $(m - \beta)$ higher order bits of i and j are row and column indices of S_M within M respectively and the β lower order bits of i and j are the row and column indices of e within S_M. That is, if we write $i = (i_{m-1}i_{m-2}...i_\beta i_{\beta-1}i_{\beta-2}...i_1i_0)_2$ and $j = (j_{m-1}j_{m-2}...j_\beta j_{\beta-1}j_{\beta-2}...j_1j_0)_2$, the Morton-hybrid index z_Θ is given by

$$z_\Theta = (i_{m-1}j_{m-1}i_{m-2}j_{m-2}...i_{\beta+1}j\beta + 1i_\beta j_\beta i_{\beta-1} \times i_{\beta-2}...i_1i_0j_{\beta-1}j_{\beta-2}...j_1j_0)_2.$$

3.4.1. Encoding in the Morton-hybrid order

To encode a Cartesian index (i, j) in the Morton-hybrid order, we proceed as follows. The last β bits of i must be extracted. To do this, mask i with $\mu = (1 \ll \beta) - 1$ to extract i_r, the part of the binary representation of i representing the row offset of the element within the row-major block. Thus $i_r = i\&\mu = (i_{\beta-1}i_{\beta-2}...i_1i_0)_2$. The rest of i is denoted by i_m. It represents the row index of the row-major block in which the element (i, j) lies. The value of i_m is given by

$$i_m = i\&(0xFFFFFFFF \ll \beta) = (i_{m-1}...i_{\beta+1}i_\beta 0000..)_2.$$

We find $j_r = (j_{\beta-1}j_{\beta-2}...j_1j_0)_2$ and $j_m = (j_{m-1}...j_{\beta+1}j_\beta 0000..)_2$, the row-major and Morton parts of the index j respectively in a similar manner. The values j_r and j_m are the column offset of the element in the row-major block and the column index of this block respectively. The Morton-hybrid index z_Θ is found by dilating i_m and j_m into $i'_m = (0i_{m-1}...0i_{\beta+1}0i_\beta 00000000)_2$ and $j'_m = (0j_{m-1}...0j_{\beta+1}0j_\beta 00000000)_2$ respectively, using Alg. 1. Then, z_Θ is given by:

$$z_\Theta = ((i'_m \ll 1)|j'_m)|(i_r \ll \beta)|j_r. \tag{6}$$

Note that this equation is in fact made of two parts: $(i'_m \ll 1)|j'_m$ performs the interleaving of the Morton parts of the indices as from Eq. (5) from Section 3.3, and $(i_r \ll \beta)|j_r$ serves to append the row-major parts of the indices with $m = \beta$. To verify that this results in z_Θ as given in Eq. (1), we perform:

$$(i'_m \ll 1) = (i_{m-1}0...i_{\beta+1}0i_\beta 00...00)_2,$$

$$j'_m = (0j_{m-1}...0j_{\beta+1}0j_\beta 00...00)_2,$$

$$(i_r \ll \beta) = (i_{\beta-1}i_{\beta-2}...i_1i_00...0)_2,$$

and

$$j_r = (j_{\beta-1}j_{\beta-2}...j_1j_0)_2.$$

We perform OR operations on the above values to get

$$z_\Theta = (i_{m-1}j_{m-1}...i_{\beta+1}j_{\beta+1}i_\beta j_\beta i_{\beta-1}i_{\beta-2}...i_1i_0j_{\beta-1}j_{\beta-2}...j_1j_0)_2.$$

3.4.2. Decoding in the Morton-hybrid order

When decoding an index in the Morton-hybrid order, we start with

$$z_\Theta = (i_{m-1}j_{m-1}...i_{\beta+1}j_{\beta+1}i_\beta j_\beta i_{\beta-1}i_{\beta-2}...i_1i_0j_{\beta-1}j_{\beta-2}..j_1j_0)_2$$

We wish to find the Cartesian index (i, j) by isolating the bits of i and the bits of j. Recall that z_Θ is in fact made of three parts as described in the introduction of this section: the Morton index of the row-major block in which this element lies, the row offset of the element within this block, and the column offset of the element within this block. Thus to determine i, the Morton index of the row-major block must be decoded to get a row index, and the row offset of the element within the block is concatenated to this to get i. The same applies for j. We extract the two parts for i and j from above as follows, in a reverse process to encoding. Recall from the encoding procedure the intermediate values i_r, i_m, i'_m, j_r, and j_m. We mask z_Θ by $((0xAAAAAAAA \ll \beta) \ll \beta)$ to get i'_m and mask z_Θ by $((0x55555555 \ll \beta) \ll \beta)$ to get j'_m. The result is $i'_m = (i_{m-1}0...i_{\beta+1}0i_\beta 000....00)_2$ and $j'_m = (0j_{m-1}...0j_{\beta+1}0j_\beta 00...00)_2$. Then, $(i'_m \gg 1)$ and j'_m are un-dilated using Alg. 2 to get $i_m = (i_{m-1}...i_{\beta+1}i_\beta 0...0)_2$ and $j_m = (j_{m-1}..j_{\beta+1}j_\beta 0...0)_2$ respectively. To extract i_r and j_r, we mask z_Θ with $(\mu \ll \beta)$ and μ respectively to get $i_r = (i_{\beta-1}i_{\beta-2}...i_1i_00...0)_2$ and $j_r = (j_{\beta-1}j_{\beta-2}..j_1j_0)_2$. The values for i and j can then be found as $i = i_m|(i_r \gg \beta)$ and $j = j_m|j_r$, resulting in $i = (i_{m-1}...i_{\beta+1}i_\beta i_{\beta-1}...i_1i_0)_2$ and $i = (j_{m-1}...j_{\beta+1}j_\beta j_{\beta-1}...j_1j_0)_2$.

4. Performance analysis

4.1. Computation overhead for the Hilbert order

Each of encoding and decoding requires m iterations. In each iteration, the encoding operation uses six bit operations and two table look-ups and the decoding algorithm uses eight bit operations and two table look-ups. The first of each table look-up incurs a random cache miss. The tables are small enough to fit in internal memory. As row and column permutations swaps in TURBO take place consecutively in one batch, so do the conversion routines, each of which requires access to the look-up table. By the LRU cache policy, the tables are kept in internal memory until the permutations have concluded. Consequently, the overall I/O cost for table lookups is $\Theta(1)$ per one batch of permutations, which is dominated by the I/O cost of the matrix operations in each recursive step, and hence, can be discarded. Summarising, the overall run-time for each of encoding and decoding in the Hilbert order is $\Theta(m)$ bit operations.

Algorithms for encoding and decoding in the Hilbert order which do not use look-up tables are described in full detail in [16]. In [8], a new variant of the Hilbert curve is introduced for which the encoding runtime overhead is $O(\lg(\max(i, j)))$. Yet, such conversion schemes are still beaten by the Morton and Morton-hybrid layouts.

4.2. Computation overhead for the Peano order

The costs associated with the coding of the Peano order are affected by base 3 computations. As described in Section 3.2, the coding procedure results in m iterations with a constant number of operations in base 3 per iteration, which cannot be replaced by bit operations. Conversion between base 3 and base 2 values is also needed at the end of the iterative computations in order to get the final indices. Despite that the coding costs are $\Theta(m)$, the constants governing this asymptotic relationship are higher than with the Hilbert, Morton, and Morton-hybrid orders. As such, we rule out using the Peano order within the TURBO algorithm that consistently requires row-wise and column-wise traversal of the matrix for pivoting and permutations — in contrast to matrix multiplication where any traversal that suits the layout may be used.

Table 6
Coding overhead.

Curve	Encoding costs	Decoding costs
Hilbert	$\Theta(m)$ operations in base 2	$\Theta(m)$ operations in base 2
Peano	$\Theta(m)$ operations in base 3	$\Theta(m)$ operations in base 3
Morton	26 operations in base 2	26 operations in base 2
Morton-hybrid	36 operations in base 2	38 operations in base 2

4.3. Computation overhead for the Morton order

Assume that the input matrix has dimensions $2^\alpha \times 2^\alpha$, where α is the machine word-size. This guarantees that each Cartesian index fits in a machine word. For the typical value $\alpha = 64$, such matrix sizes are very generous. Each of dilation and un-dilation performs twelve bit operations: four OR operations, four AND operations, and four bit-shift operations. The encoding procedure makes use of the dilation algorithm twice, followed by one bit-shift operation and one OR operation. This results in a total of twenty six bit operations for encoding an index in the Morton order. The decoding procedure makes use of two masking operations, one bit-shift operation, and two calls to the un-dilation algorithm: one for un-dilating i_z and one for un-dilating j_z. This requires a total of twenty seven bit operations for decoding a Morton index to a Cartesian index.

4.4. Computation overhead for the Morton-hybrid order

As above, assume the input matrix has dimensions $2^\alpha \times 2^\alpha$, where α is the machine word-size. The encoding algorithm for the Morton-hybrid layout performs the same twenty six bitwise operations as those for the encoding algorithm for the Morton layout for bit dilation, with an additional ten operations: four AND operations, four bit shift operations, and two OR operations. Thus encoding in the Morton-hybrid order costs thirty six bit operations. The decoding algorithm for the Morton-hybrid layout performs the same twenty seven operations used in the decoding procedure for the Morton order. In addition to those, seven bit shift operations, two AND operations, and two OR operations are needed, totalling to eleven additional bit operations. Thus decoding Morton-hybrid index to get the Cartesian index requires thirty eight bit operations.

4.5. Summary

Given a $2^m \times 2^m$ matrix, Table 6 summarises the coding overheads for each of the curves considered:

4.6. Experimental results

We implemented the four coding variants in the C language using a 64-bit Intel Pentium IV machine with processor speed of 3 GHz. The Hilbert processes implemented follow the state-of-the-art method in [17]. Comparison of the running times as a function of the number of cycles of each of encoding and decoding are reported in Tables 7–10. We consider matrices of sizes corresponding to those where the associated TU decomposition has been tested using the Morton-hybrid order in [1] achieving orders of magnitudes reduction in cache misses and overall run-time. For all of Hilbert, Morton, and Morton-hybrid orders, the input matrices are of dimension $2^m \times 2^m$. For the Peano order, the input matrices are of dimension $3^m \times 3^m$. Our results capture the real life performance of the various coding (encoding and decoding) processes whose theoretical performance in the previous section has been detailed. In contrast to the theoretical scenario which assumes that all operands are in CPU registers, this assumption

Table 7
Coding overhead for the Peano order.

m	Encoding	Decoding
5	421	731
6	483	870
7	552	1008
8	620	1147

Table 8
Coding overhead for the Morton order.

m	Encoding	Decoding
8	291	510
10	351	632
12	433	801
13	576	996

Table 9
Coding overhead for the Morton order.

m	Encoding	Decoding
8	143	148
10	173	175
12	165	184
13	198	212

Table 10
Coding overhead for the Morton-hybrid order.

m	Encoding	Decoding
8	190	198
10	203	200
12	221	226
13	217	219

is almost always violated. Also, in a real-life application, a certain piece of code will feature contention among various instructions for the same parts of the CPU core, and when hyperthreading is on, a certain core executes operations from two threads. All of this renders throughput somehow more deviant in part from the theoretical analyses, as performance becomes strongly dependent on what neighbouring threads are doing. Still, our empirical results below confirm the insight gathered from our theoretical section as follows. We observe that the run-times of both the Peano and Hilbert based coding processes increase with the matrix dimensions, where the coding in the Peano order requiring more time as a result of the computations in base 3. Morton and Morton-hybrid run-times appear to be stagnating and of the same order of growth, with the Morton-hybrid appearing to be slightly more expensive.

5. Security analysis and concluding remarks

The RSA algorithm has become the de facto standard for encryption, especially for data sent over the internet. RSA takes its security from the hardness of the Integer Factorisation problem or solving the discrete logarithm for a composite moduli. Both of these problems require solving large systems of linear equations over finite fields [23,25]. As the size of the modulus of an RSA key grows with the number of bytes to be encrypted, the corresponding linear system to be solved in the adversary integer factorisation algorithms like the quadratic sieve and multiple quadratic sieve, or the discrete logarithm algorithms, also grows. With the proliferation of yet bigger data that is pushing mathematical software to more extremes, RSA adversaries would attempt to redesign linear solvers over finite fields to have them exploit the memory hierarchy. In return for this, RSA security mechanisms would have to adapt in anticipation of such adversarial attempts made possible via exploiting the memory hierarchy.

Our present work is a precursor for a cache oblivious implementation of the TU decomposition, a direct linear solver over finite fields, using the Morton hybrid order. In this paper we have reviewed four major space-filling curve representations as they apply to parallel TU decomposition over finite fields (TURBO). Whilst these representations have been traditionally employed to develop cache-oblivious matrix multiplication and factorisation algorithms, both serial and parallel, we find that they incur additional costs associated with encoding and decoding from the row major layout as needed for the row and column permutations within TURBO. Our detailed analysis of the bit operations required, and in some cases the number of table look-ups, shows that the Morton and Morton-hybrid order are the best candidates. Additionally, the Morton-hybrid order balances this cost with that of recursion overhead and thus is a better candidate than the purely Morton order. The resulting TU algorithm obtained in [1] is inherently nested-parallel, and has low span, for which the natural sequential evaluation order has lower cache miss rate, and attains run-time speedups over 70% for large matrices.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jpdc.2019.12.016.

References

[1] F.K. Abu Salem, M. Al Arab, Cache-oblivious matrix multiplication for exact TU factorisation, in: Proceedings of PASCO '17, ACM Press, 2017, pp. 1–13.
[2] M.D. Adams, D.S. Wise, Fast additions on masked integers, SIGPLAN Not. 41 (5) (2006) 39–45.
[3] M. Bader, C. Mayer, Cache Oblivious Matrix Operations Using Peano Curves, in: Proceedings of PARA '06, in: Lecture Notes in Computer Science, Vol. 4699, 2007, pp. 521–530.
[4] M. Bader, C. Zenger, A Cache oblivious algorithm for matrix multiplication based on peano's space filling curve, in: Proceedings of PPAM '05, in: Lecture Notes in Computer Science, vol. 3911, 2005, pp. 1042–1049.
[5] G. Blelloch, P.B. Gibbons, H.-V. Simhadri, Low depth cache-oblivious algorithms, in: Proceedings of SPAA 2010, ACM Press, 2010, pp. 189–199.
[6] S. Chatterjee, A.R. Lebeck, P.K. Patnala, M. Thottethodi, Recursive array layouts and fast parallel matrix multiplication, in: Proceedings of SPAA '99, ACM Press, 1999, pp. 222–231.
[7] S. Chatterjee, A.R. Lebeck, M. Thottethodi, P.K. Patnala, Recursive array layouts and fast matrix multiplication, in: Proceedings of SPAA '99, ACM Press, 1999, pp. 1105–1123.
[8] N. Chen, N. Wang, B. Shi, A new algorithm for encoding and decoding the hilbert order, Softw. Pract. Exp. 37 (8) (2007) 897–908.
[9] J.G. Dumas, J.L. Roche, A parallel block algorithm for exact triangulizations, Parallel Comp. 28 (11) (2002) 1531–1548.
[10] A.J. Fisher, A new algorithm for generating Hilbert curves, Softw. - Pract. Exp. 16 (1) (1986) 5–12.
[11] J.D. Frens, D.S. Wise, QR Factorization with morton-ordered quadtree matrices for memory re-use and parallelism, in: Proceedings of PPoPP '03, ACM Press, 2003, pp. 144–154.
[12] A. Heinecke, M. Bader, Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms, in: Proceedings of MAW '08, ACM Press, 2008, pp. 385–392.
[13] D. Hilbert, Über Die stetige abbildung einer linie auf ein flächenstück, Math. Ann. 38 (1891) 459–460.
[14] O.H. Ibarra, S. Moran, R. Hui, A generalization of the fast LUP matrix decomposition algorithm and applications, J. Algorithms 3 (1) (1982) 45–56.
[15] O.H. Ibarra, S. Moran, L.E. Rosier, A note on the parallel complexity of computing the rank of order n matrices, Inf. Process. Lett. 11 (4–5) (1980).
[16] X. Liu, G. Schrack, Encoding and decoding the Hilbert order, Softw. Pract. Exp. 26 (12) (1996) 1335–1346.
[17] X. Liu, G. Schrack, An algorithm for encoding and decoding the 3-d Hilbert order, IEEE Trans. Image Process. 6 (9) (1997) 1333—1337.
[18] X. Liu, G. Schrack, A new ordering strategy applied to spatial data processing, Int. J. Geogr. Inf. Sci. 12 (1) (1998) 3–22.

[19] K. Matsumoto, N. Nakasato, T. Sakaia, H. Yahagi, S. G.Sedukhina, Multi-level optimization of matrix multiplication for GPU-equipped systems, Procedia Comput. Sci. 4 (2011) 342–351.
[20] P. Merkey, Z-ordering and UPC, Technical Report, Michigan Technological University, Houghton, Michigan, 2003.
[21] Y. Moisan, Spark performance tuning from the trenches, 2018, URL https://medium.com/teads-engineering/spark-performance-tuning-from-the-trenches-7cbde521cf60.
[22] G.M. Morton, A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Technical Report, IBM Ltd, Ottawa, Canada, 1966.
[23] A.M. Odlyzko, Discrete logarithms in finite fields and their cryptographic significance, in: Advances in Cryptology: Proceedings of Eurocrypt '84, in: Lecture Notes in Computer Science, vol. 209, 1985, pp. 224–314.
[24] G. Peano, Sur une courbe qui remplit toute une air plaine, Math. Ann. 36 (1890) 157–160.
[25] C. Pomerance, Analysis and comparison of some integer factoring algorithms, in: Computational Methods in Number Theory: Part I, Vol. 154, Math. Centre Tract, 1982, pp. 89–139.
[26] R. Raman, D.S. Wise, Converting to and from dilated integers, IEEE Trans. Comput. 57 (4) (2008) 567–573.
[27] D.S. Wise, C.L. Citro, J.J. Hursey, F. Liu, M.A. Rainey, A paradigm for parallel matrix algorithms: Scalable Cholesky, in: Proceedings of Euro-Par '05, in: Lecture Notes in Computer Science, vol. 3648, 2005, pp. 687–698.

Fatima K. Abu Salem received the B.S. degree and M.S. degree in Mathematics from the American University of Beirut, Beirut, Lebanon, and the Ph.D. degree in Computing Science from The University of Oxford, Oxford, England. She is an Associate Professor at American University of Beirut, Beirut, Lebanon. Her research interests include Algorithm Engineering in Computer Algebra, Cache-efficient and Parallel Algorithms, as well as Data Science for the social good.

Mira Al Arab received her B.S. and M.S. degrees in Computer Science from the American University of Beirut. She currently is associate manager of software development at Vilo.ai, Lebanon.

Laurence Tianruo Yang received the B.E. degree in Computer Science and Technology from Tsinghua University, Beijing, China, and the Ph.D. degree in computer science from the University of Victoria, Victoria, BC, Canada. He is currently a Professor at St. Francis Xavier University, Antigonish, NS, Canada. His research interests include Parallel and Distributed Computing, Embedded and Ubiquitous/Pervasive Computing, and Big Data.