Performance Improvement on k^2-Raster Compact Data Structure for Hyperspectral Scenes

Kevin Chow, Dion Eustathios Olivier Tzarmarias, Miguel Hernández-Cabronero, Ian Blanes, *Senior Member, IEEE*, and Joan Serra-Sagristà, *Senior Member, IEEE*

***Abstract*— This letter proposes methods to improve data size and access time for k^2-raster, a losslessly compressed data structure that provides efficient storage and real-time processing. Hyperspectral scenes from real missions are used as our testing data. In previous studies, with k^2-raster, the size of the hyperspectral data was reduced by up to 52% compared with the uncompressed data. In this letter, we continue to explore novel ways of further reducing the data size and access time. First, we examine the possibility of using the raster matrix of hyperspectral data without any padding (unpadded matrix) while still being able to compress the structure and access the data. Second, we examine some integer encoders, more specifically the Simple family. We discuss their ability to provide random element access and compare them with directly addressable codes (DACs), the integer encoder used in the original description for k^2-raster. Experiments show that the use of unpadded matrices has improved the storage size up to 6% while the use of a different integer encoder reduces the storage size up to 6% and element access time up to 20%.**

***Index Terms*— Directly addressable codes (DACs), image compression, lossless hyperspectral imaging, PForDelta, remote sensing, Simple-9, Simple-16.**

I. Introduction

Hyperspectral scenes are data taken from the air by sensors, such as airborne visible/infrared imaging spectrometer (AVIRIS), or from space by satellite instruments such as Hyperion and infrared atmospheric sounding interferometer (IASI). These scenes are made up of multiple bands from across the electromagnetic spectrum, and data extracted from certain bands have many practical applications, such as oil field exploration and mineral exploration. Due to their relatively large sizes, hyperspectral scenes are usually compressed to increase transmission throughput and reduce data volumes.

Compact data structures can store data efficiently and provide real-time data compression and access to the original data [1]. They can be loaded into main memory, and operations to access data are often carried out by means of the rank and select functions [2]. Compact data structures provide reduced space usage and query time. There is no need to decompress a large portion of the structure to access and query individual data as it is the case with data compressed by classical compression algorithms such as gzip and specialized algorithms such as CCSDS 123.0-B-2 [3].

In this work, we reduce the hyperspectral data size using k^2-raster, a compact data structure, to produce lossless compression. In our previous letter [4], we presented a predictive method and a differential method that made use of spatial and spectral correlations in hyperspectral data with favorable results. Nevertheless, due to the nature of these methods, only random access to individual cells can be done, whereas other operations such as query on a region cannot be performed. In this letter, we focus on investigating whether unpadded matrices and variable-length integer encoders other than directly addressable codes (DACs) [5] can provide competitive compression ratios as well while improving random and query access time. In our case, we need to store non-negative small integers in the k^2-raster tree structure, which is built in such a way that the nodes are not connected by pointers but can still be reached with the use of a compact data structure's linear rank function. Fig. 1 depicts a global picture of the interrelations between the elements discussed above. The compact data structures that we have been working on are still at the research stage when applied to hyperspectral scenes, but with the encouraging results that we have obtained so far, we can extrapolate their practical use in applications of remote sensing and geographic information systems [6]–[10].

The letter is organized as follows. Section II provides background information on k^2-raster built from a padded matrix and the various integer encoders. Section III describes the proposed method of using an unpadded matrix to build the structure and introduces the different variable-length integer encoders. Section IV presents some experimental results. Section V sums up the key points of our discussion.

II. Background

Ladra *et al.* [11] proposed k^2-raster, a tree structure specifically designed for raster data including images. It is built from a matrix of width w and height h, and an integer $k \geq 2$. If the matrix can be partitioned into k^2 square quadrants of equal size, it can be used directly. Otherwise, it is necessary to enlarge the matrix to size $s \times s$, where $s = k^{\lceil \log_k \max(w,\ h) \rceil}$, and the number of subdivisions is $\log_k(s)$. The padding elements are equal to zero. This extended (padded) matrix is then recursively partitioned into k^2 square submatrices of identical size, hereafter referred to as quadrants. This process is repeated

Manuscript received January 15, 2021; revised April 29, 2021; accepted May 17, 2021. Date of publication June 9, 2021; date of current version December 28, 2021. This work was supported in part by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund (Programa Formación de Personal Investigador) under Grant RTI2018-095287-B-I00 and Grant BES-2016-078369, in part by the Catalan Government under Grant 2017SGR-463, in part by the Postdoctoral Fellowship Program Beatriu de Pinós funded by the Secretary of Universities and Research (Government of Catalonia) under Grant 2018-BP-00008, and in part by the Horizon 2020 Program of Research and Innovation of the European Union under the Marie Skłodowska-Curie Grant Agreement 801370. *(Corresponding author: Kevin Chow.)*

The authors are with the Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain (e-mail: kevin.chow@uab.cat).

Digital Object Identifier 10.1109/LGRS.2021.3084065

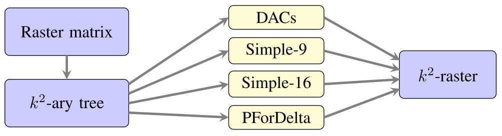


Fig. 1. Construction of k^2-raster. A k^2-ary tree is first built from a raster matrix. Compression and random access are achieved when tree node data are encoded by an integer encoder, such as DACs, Simple-9, Simple-16, or PForDelta, resulting in a k^2-raster structure.

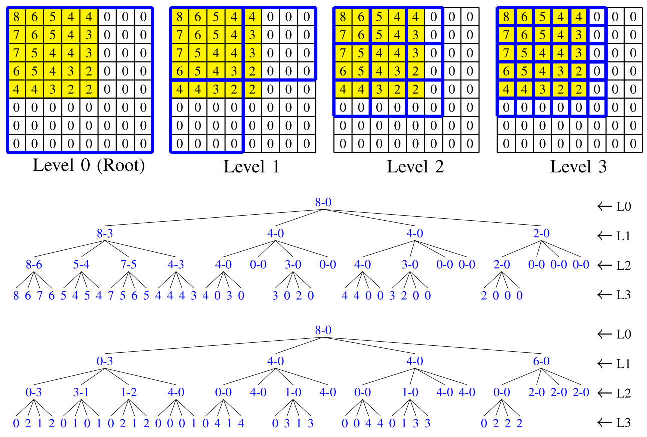
until all cells in a quadrant have the same value, or until the submatrix has size 1×1 and cannot be further subdivided. This partitioning produces the nodes for a k^2-ary tree topology where the data in the nodes are stored in the following data structures.

1) At each tree level ℓ, the maximum and minimum values of each quadrant are computed. These are then compared with the corresponding maximum and minimum values in the parent node, and the differences are stored in the $V_{\max_\ell}$ and $V_{\min_\ell}$ arrays of each level. Saving the differences instead of the original values results in smaller values for each node, which in turn allows a better compression with an integer encoder. Next, with the exception of the root node at the top level, the $V_{\max_\ell}$ and $V_{\min_\ell}$ arrays at all the levels are concatenated to form $L_{\max}$ and $L_{\min}$, respectively. Both arrays are then compressed by an integer encoder.
2) The root's maximum (rMax) and minimum (rMin) values are stored as uncompressed integers.
3) A bitmap array T is generated from all the nodes except the ones at the root and at the last level, each node indicating whether it has child nodes or not. This bitmap serves to locate the tree nodes when cell queries are performed by means of a rank function [2].

In Fig. 2, an example of a 5×5 matrix is shown to illustrate this process. The elements which fully describe the resulting k^2-raster structure are shown at the bottom of Fig. 2. Please refer to [12] for a more comprehensive description of k^2-raster.

DACs were proposed by Brisaboa *et al.* [5]. Consider a sequence of integers x. Each integer x_i, which is represented by $\lfloor \log_2 x_i \rfloor + 1$ bits, is broken into chunks of bits of size C_S. Each chunk is stored in a block of size $C_S + 1$ with the additional (highest) bit used as a control bit (0 for most significant bits, 1 otherwise). More information on DACs and software implementation can be found in the paper by Brisaboa *et al.* [5].

Simple-9 word-aligned encoding [13] is another approach to compression. Each 32-bit word is split into two parts: a 28-bit part where a variable number of integers are encoded and a 4-bit part which is a selector with a value ranging from 0 to 8. For example, selector 0 signals that the first 28 integers in the data have a value of 0 or 1. Selector 1 signals that it can pack 14 integers into the segment with a maximum bit length of 2 bits for each. Beginning with selector 0, each selector is tested until the most efficient one is found. Simple-16 [14] is a variant of Simple-9 and uses all the 16 combinations in the selector bits. Their values range from 0 to 15. PForDelta [15] is also similar to both Simple-9 and Simple-16 but encodes a fixed group of 32, 64, 128, or 256 integers in a varying number of bytes. A predetermined percentage of those numbers that are larger than the others are encoded separately after the smaller numbers or in another location.



Element	Bin/Dec	Tree Level	Node Data
L_{max}	Decimal	Level 1	0446
		Level 2	0314 0414 0144 0222
		Level 3	0212 0101 0212 0001 0414 0313 0044 0133 0222
L_{min}	Decimal	Level 1	3000
		Level 2	3120 00 00 0
$rMax$	Decimal		8
$rMin$	Decimal		0
T Bitmap	Binary		1111 1111 1010 1100 1000

Fig. 2. Top: A 5×5 matrix example showing recursive partitioning. Middle: The upper tree is a k^2-raster ($k = 2$) tree constructed from the matrix and the lower tree takes into account the differences between the parent and child nodes. Bottom: A table showing the elements of the k^2-raster with padding.

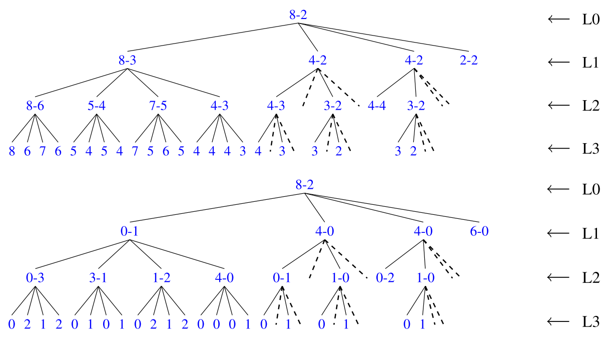
III. Proposal

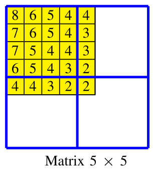
A. Proposed Building k^2-Raster Without Padding

As mentioned in Section II, given a k value, a matrix with a size that is not a power of k^2 needs to be extended according to the equation for computing s in that section. The values of the new cells are set to zero. This is necessary because the search from the root down to its leaves requires the knowledge of its child node location using the rank function, which is a function of the number of child nodes each parent node has. Adding new cells, however, also means that the nodes that are outside the matrix have to be saved, and this leads to a larger structure.

To illustrate the above point, we construct a tree based on the elements within the bounds of the original matrix where $k = 2$. This matrix together with its corresponding k^2-raster tree are shown in Fig. 3. This is done without padding and is therefore not a full quaternary tree (a full quaternary tree is one where each node has either 0 or 4 child nodes). In this case, the parent does not know how many children it has, so it is not possible to use the rank function to get to the correct child nodes without including them in the T bitmap. On the other hand, with padding, as explained earlier in Section II, it is a full quaternary tree. Fig. 2 shows the padded matrix and the tree that is built from it. Comparing it with Fig. 3, we can see that the k^2-raster tree from the unpadded matrix can save fewer elements.

To build a non-full quaternary tree, we modify the original function for building the tree and prune the values that are located outside the bounds of the original matrix. This reduces the number of $L_{\max}$ and $L_{\min}$ values that need to be saved. However, when forming the T bitmap, the full-quaternary tree is still used to ensure that the parent nodes can reach their child nodes, with the corresponding bit value of the node that





Element	**Bin/Dec**	**Tree Level**	**Node Data**
T Bitmap	Binary		1110 1111 1010 0100
L_{max}	Decimal	Level 1	0446
		Level 2	0314 01 01
		Level 3	0212 0101 0212 0001 01 01 01
L_{min}	Decimal	Level 1	100
		Level 2	3120 10 0
$rMax$	Decimal		8
$rMin$	Decimal		2

Fig. 3. Top: A non-full quaternary tree constructed from a 5 × 5 matrix without padding. The second and third parent nodes at Level 1 have only two child nodes each. All nodes that are outside the bounds are connected by dashed lines and will not be saved, and the corresponding value in T bitmap is set to zero. Bottom: The 5 × 5 matrix and a table showing the elements of the k^2-raster without padding.

is outside the bounds being set to zero. This facilitates the search path through the full quaternary tree when cell queries are performed using the rank function. Note that the T bitmap does not have to store the location information of the nodes of the last level because we can compute the location of the values from the original matrix size and the s value in the expanded matrix.

The algorithms for building k^2-raster with and without padding are listed in Algorithm 1. What this build function does is it excludes elements that are outside the bounds of the original matrix and save only the actual data. This helps reduce the size of the structure. To compute the theoretical storage reduction, we can count how many symbols 0 we are sparing in the encoding with the unpadded matrix: spared0. Since the actual reduction in storage depends on the entropy encoding, we could estimate the saving to be spared0 × $\log_2$(Probability(spared0)). Another way to estimate space saving is to calculate the size of the original raster matrix originalDim compared with that of the expanded matrix expandedDim. The maximum saving can be estimated to be ((expandedDim − originalDim)/expandedDim) × 100% but the saving is, in general, less, due to factors such as the k value, tree height, and k^2-raster saving such as nodes at a higher level that become leaves. For example, in Figs. 2 and 3, the estimated maximum saving is ((64 − 25)/64) × 100% = 61% and the $L_{\max}$ nodes' saving is actually ((56 − 34)/56) × 100% = 39%. Hence, there is a relationship between the image dimensions and the storage saving.

B. Random Access for Integer Encoders

In saving the $L_{\max}$ and $L_{\min}$ arrays, the authors of k^2-raster made use of DACs as an integer encoder for random access to its elements. In this research, we have investigated other word-aligned integer encoders from the Simple family: Simple-9 and Simple-16, and the PForDelta variant, which also allow random access due to their structure.

Algorithm 1: Algorithm for Constructing $T, Vmax, Vmin$ for a Padded Matrix and an Unpadded Matrix. With the "+" Lines Removed, the Pseudocode Is for the Function **Build**$(n, nx, ny, l, r, c, o_l)$ for Building From an Unpadded Matrix M. With the "−" Lines Removed From the Code and the "+" Lines Re-Added, It Becomes One for the Function **Build**(n, l, r, c) From a Padded Matrix M

```
  $maxval \leftarrow 0$
  $minval \leftarrow \infty$
- $outcount \leftarrow 0$
  for $i \leftarrow 0 \ldots k-1$ do
      for $j \leftarrow 0 \ldots k-1$ do
          if $n = k$ then /* last level */
-             if $(r+i) < nx$ and $(c+j) < ny$ then
-                 $o_l[pmax_l] \leftarrow 0$
-             else
-                 $o_l[pmax_l] \leftarrow 1$
-             $outcount \leftarrow outcount + o_l[pmax_l]$
              $maxval \leftarrow max(maxval, M_{r+i,c+j})$
              $minval \leftarrow min(minval, M_{r+i,c+j})$
              $Vmax_l[pmax_l] \leftarrow M_{r+i,c+j}$
              $pmax_l \leftarrow pmax_l + 1$
          else /* in-between level */
-             if $(r + i \cdot (n/k)) < nx$ and $(c + j \cdot (n/k)) < ny$
               then
-                 $o_l[pmax_l] \leftarrow 0$
-             else
-                 $o_l[pmax_l] \leftarrow 1$
-             $outcount \leftarrow outcount + o_l[pmax_l]$
-             $(childmax, childmin, childoutcount) \leftarrow$
               Build$(n/k, nx, ny, l+1, r + i \cdot (n/k), c + j \cdot$
               $(n/k), o_l)$
+             $(childmax, childmin) \leftarrow$
               Build$(n/k, l+1, r + i \cdot (n/k), c + j \cdot (n/k))$
              $Vmax_l[pmax_l] \leftarrow childmax$
              $Vmin_l[pmin_l] \leftarrow childmin$
-             if
               $childoutcount = k^2$ or $childmax = childmin$
               then
+             if $childmax = childmin$ then
                  $T_l[pmax_l] \leftarrow 0$
              else
                  $T_l[pmax_l] \leftarrow 1$
              $pmax_l \leftarrow pmax_l + 1$
              $pmin_l \leftarrow pmin_l + 1$
              $maxval \leftarrow max(maxval, childmax)$
              $minval \leftarrow min(minval, childmin)$
+ if $maxval = minval$ then
+     $pmax_l \leftarrow pmax_l - k^2$
+     $pmin_l \leftarrow pmin_l - k^2$
+ return $(maxval, minval)$
- return $(maxval, minval, outcount)$
```

The use of partial sums and sampling described in [1, §3.3 and §4.2] can be used in the Simple family of codes to expedite the search process in the compressed array. We incorporate such strategies by sampling these arrays at

TABLE I

TESTING SCENE DATA USED AND A COMPARISON OF k^2-RASTER STORAGE SIZE WITH PADDED MATRIX AND UNPADDED MATRIX. SIZE REDUCTIONS (%) FROM PADDED MATRIX TO UNPADDED MATRIX ARE ENCLOSED IN PARENTHESES. IN THE DIMENSIONS COLUMN x IS THE IMAGE WIDTH, y THE IMAGE HEIGHT, AND z THE NUMBER OF SPECTRAL BANDS. THE BEST RESULTS ARE HIGHLIGHTED.

Sensor	Name	Acro-nym	Original Dimensions (x × y × z)	Gzip Rate (bpppb)	Best k Value	Padded Matrix (bpppb)							Unpadded Matrix (bpppb)						
						DACs	S9*	S16*	PFD* 32	PFD* 64	PFD* 128	PFD* 256	DACs	S9*	S16*	PFD* 32	PFD* 64	PFD* 128	PFD* 256
AIRS	9†	AG9	90×135×1501	10.16	6	**9.49**	10.06	9.69	9.88	9.59	9.59	9.82	8.94 (5.8)	9.44 (6.1)	9.08 (6.3)	9.28 (6.1)	8.94 (6.8)	**8.90** (7.2)	9.07 (7.6)
	16†	AG16	90×135×1501	9.82	6	**9.12**	9.64	9.30	9.55	9.20	9.15	9.44	8.60 (5.7)	9.02 (6.4)	8.68 (6.7)	8.95 (6.2)	8.55 (7.0)	**8.45** (7.7)	8.53 (9.6)
	60†	AG60	90×135×1501	10.53	6	9.81	10.50	10.12	10.19	9.82	**9.75**	10.00	9.28 (5.4)	9.89 (5.7)	9.51 (5.9)	9.65 (5.3)	9.22 (6.1)	**9.10** (6.7)	9.19 (8.1)
	126†	AG126	90×135×1501	10.33	6	9.61	10.25	9.81	9.98	9.61	**9.53**	9.84	9.07 (5.6)	9.65 (5.9)	9.21 (6.2)	9.41 (5.7)	9.00 (6.4)	**8.85** (7.1)	9.01 (8.4)
	129†	AG129	90×135×1501	9.50	6	8.65	9.01	**8.61**	9.01	8.69	8.67	8.95	8.10 (6.3)	8.39 (7.0)	7.98 (7.3)	8.39 (6.9)	8.01 (7.8)	**7.92** (8.7)	7.98 (10.9)
	151†	AG151	90×135×1501	10.31	6	9.53	9.99	9.54	9.79	9.43	**9.41**	9.79	8.97 (5.8)	9.39 (6.0)	8.94 (6.3)	9.16 (6.4)	8.78 (6.9)	**8.71** (7.4)	8.84 (9.7)
	182†	AG182	90×135×1501	10.64	6	**9.68**	10.44	10.01	10.09	9.79	9.77	10.08	9.14 (5.6)	9.86 (5.6)	9.42 (5.9)	9.53 (5.6)	9.20 (6.1)	**9.11** (6.8)	9.21 (8.6)
	193†	AG193	90×135×1501	10.15	6	**9.44**	10.06	9.65	9.93	9.56	9.46	9.74	8.90 (5.7)	9.45 (6.1)	9.03 (6.4)	9.33 (6.0)	8.89 (7.0)	**8.72** (7.8)	8.80 (9.7)
	Average			10.18		**9.42**	9.99	9.59	9.80	9.46	**9.42**	9.71	8.88 (5.7)	9.39 (6.1)	8.98 (6.4)	9.21 (6.0)	8.82 (6.7)	**8.72** (7.4)	8.83 (9.1)
AVIRIS	Yellowstone sc. 00‡	AC00	677×512×224	10.12	6	9.61	10.37	10.11	9.80	9.41	**9.35**	9.40	9.47 (1.5)	10.20 (1.6)	9.95 (1.6)	9.67 (1.3)	9.28 (1.4)	**9.21** (1.5)	9.26 (1.5)
	Yellowstone sc. 03‡	AC03	677×512×224	9.59	6	9.42	9.80	9.57	9.40	9.03	**8.99**	9.07	9.29 (1.4)	9.65 (1.6)	9.42 (1.6)	9.28 (1.3)	8.92 (1.2)	**8.86** (1.4)	8.94 (1.4)
	Yellowstone sc. 10‡	AC10	677×512×224	7.41	6	7.62	7.34	7.18	7.44	7.06	**7.04**	7.14	7.49 (1.8)	7.20 (2.0)	7.04 (2.0)	7.31 (1.7)	6.94 (1.6)	**6.90** (1.9)	7.02 (1.7)
	Yellowstone sc. 11‡	AC11	677×512×224	9.04	6	8.81	9.32	9.09	9.02	8.65	**8.62**	8.72	8.67 (1.7)	9.17 (1.6)	8.94 (1.7)	8.90 (1.3)	8.52 (1.5)	**8.49** (1.5)	8.59 (1.5)
	Yellowstone sc. 18‡	AC18	677×512×224	10.00	6	9.78	10.52	10.28	9.84	9.47	**9.42**	9.50	9.65 (1.3)	10.37 (1.4)	10.14 (1.4)	9.73 (1.1)	9.35 (1.2)	**9.30** (1.3)	9.38 (1.3)
	Average			9.23		9.05	9.47	9.25	9.10	8.72	**8.68**	8.76	8.91 (1.5)	9.32 (1.6)	9.10 (1.6)	8.98 (1.3)	8.60 (1.4)	**8.55** (1.5)	8.63 (1.5)
	Yellowstone sc. 00†	AU00	680×512×224	12.39	9	11.92	14.01	13.79	11.93	11.54	**11.44**	11.55	11.75 (1.4)	13.83 (1.2)	13.62 (1.2)	11.75 (1.6)	11.33 (1.8)	**11.22** (1.9)	11.30 (2.2)
	Yellowstone sc. 03†	AU03	680×512×224	11.98	9	11.74	13.54	13.29	11.56	11.15	**11.08**	11.22	11.58 (1.4)	13.37 (1.3)	13.12 (1.3)	11.37 (1.6)	10.95 (1.8)	**10.87** (2.0)	10.97 (2.2)
	Yellowstone sc. 10†	AU10	680×512×224	10.17	9	9.99	10.90	10.54	9.61	9.20	**9.10**	9.26	9.82 (1.7)	10.72 (1.6)	10.36 (1.7)	9.42 (2.0)	8.98 (2.4)	**8.87** (2.6)	8.99 (2.9)
	Yellowstone sc. 11†	AU11	680×512×224	11.49	9	11.27	13.12	12.89	11.24	10.85	**10.77**	10.94	11.08 (1.7)	12.94 (1.4)	12.71 (1.4)	11.05 (1.7)	10.64 (1.9)	**10.56** (2.0)	10.69 (2.2)
	Yellowstone sc. 18†	AU18	680×512×224	12.29	9	12.15	14.19	14.01	12.10	11.70	**11.63**	11.75	11.99 (1.3)	14.01 (1.2)	13.84 (1.3)	11.91 (1.6)	11.50 (1.8)	**11.41** (1.9)	11.50 (2.1)
	Average			11.66		11.42	13.15	12.91	11.29	10.89	**10.80**	10.94	11.24 (1.5)	12.98 (1.3)	12.73 (1.4)	11.10 (1.7)	10.68 (1.9)	**10.58** (2.0)	10.69 (2.3)
CRISM	frt000065e6_07_sc164†	CR1	640×420×545	10.98	6	**10.08**	11.35	11.14	10.44	10.22	10.37	10.54	**10.00** (0.8)	11.09 (2.2)	10.89 (2.3)	10.36 (0.7)	10.14 (0.8)	10.29 (0.8)	10.45 (0.9)
	frt00008849_07_sc165†	CR2	640×450×545	11.03	6	**10.37**	11.78	11.57	10.69	10.49	10.65	10.84	**10.29** (0.8)	11.69 (0.8)	11.48 (0.8)	10.62 (0.7)	10.42 (0.7)	10.57 (0.7)	10.76 (0.7)
	frt0001077d_07_sc166†	CR3	640×480×545	11.20	6	**11.05**	12.99	12.74	11.41	11.22	11.35	11.49	**10.97** (0.7)	12.89 (0.8)	12.64 (0.8)	11.35 (0.5)	11.15 (0.7)	11.27 (0.7)	11.39 (0.8)
	hrl00004f38_07_sc181†	CR4	320×420×545	10.77	5	**9.97**	10.93	10.72	10.53	10.30	10.37	10.34	**9.90** (0.7)	10.88 (0.4)	10.67 (0.5)	10.48 (0.4)	10.24 (0.6)	10.31 (0.6)	10.28 (0.6)
	hrl0000648f_07_sc182†	CR5	320×450×545	10.90	5	**10.11**	11.24	10.99	10.67	10.47	10.53	10.50	**10.06** (0.5)	11.21 (0.2)	10.97 (0.3)	10.64 (0.3)	10.43 (0.4)	10.49 (0.4)	10.46 (0.4)
	hrl0000ba9c_07_sc183†	CR6	320×480×545	10.87	5	**10.65**	12.33	12.05	11.21	11.01	11.04	10.99	**10.57** (0.7)	12.27 (0.4)	11.99 (0.5)	11.15 (0.5)	10.95 (0.5)	10.97 (0.7)	10.92 (0.7)
	Average			10.96		**10.37**	11.77	11.54	10.82	10.62	10.72	10.78	**10.30** (0.7)	11.67 (0.8)	11.44 (0.8)	10.77 (0.5)	10.56 (0.6)	10.65 (0.7)	10.71 (0.7)
Hyperion	Agricultural‡	HC1	256×3129×242	8.84	8	8.54	9.79	9.56	8.80	8.42	**8.35**	8.37	8.52 (0.3)	9.77 (0.2)	9.54 (0.2)	8.86 (-0.7)	8.49 (-0.9)	**8.36** (-0.2)	**8.36** (0.1)
	Coral Reef‡	HC2	256×3127×242	7.45	8	7.62	8.28	7.93	7.60	7.18	**7.08**	7.10	7.62 (0.1)	8.28 (0.1)	7.93 (0.1)	7.67 (-0.8)	7.29 (-1.5)	**7.15** (-1.1)	**7.15** (-0.7)
	Urban‡	HC3	256×2905×242	8.85	8	8.86	10.30	10.04	8.91	8.51	**8.46**	8.50	8.83 (0.3)	10.28 (0.2)	10.02 (0.2)	8.93 (-0.2)	8.51 (0.0)	**8.44** (0.3)	8.48 (0.3)
	Average			8.38		8.34	9.46	9.18	8.44	8.04	**7.96**	7.99	8.32 (0.2)	9.44 (0.2)	9.16 (0.2)	8.49 (-0.6)	8.10 (-0.7)	**7.99** (-0.3)	8.00 (-0.1)
	Erta Ale†	HU1	256×3187×242	8.69	8	7.76	8.30	8.00	7.99	7.47	**7.32**	7.33	7.73 (0.5)	8.27 (0.4)	7.97 (0.4)	8.05 (-0.7)	7.56 (-1.2)	7.37 (-0.6)	**7.32** (0.2)
	Lake Montana†	HU2	256×3176×242	8.69	8	7.82	8.38	8.10	8.11	7.60	**7.46**	7.47	7.80 (0.2)	8.37 (0.1)	8.08 (0.1)	8.19 (-1.0)	7.71 (-1.3)	7.51 (-0.7)	**7.50** (-0.3)
	Mt. St. Helena†	HU3	256×3242×242	8.26	8	7.91	8.48	8.20	8.14	7.63	**7.50**	7.53	7.87 (0.5)	8.44 (0.4)	8.17 (0.5)	8.20 (-0.7)	7.74 (-1.4)	7.55 (-0.6)	**7.52** (0.1)
	Average			8.55		7.83	8.38	8.10	8.08	7.57	**7.43**	7.45	7.80 (0.4)	8.36 (0.3)	8.07 (0.3)	8.15 (-0.8)	7.67 (-1.3)	7.47 (-0.6)	**7.44** (0.0)
IASI	Level 0 1†	IASI1	60×1528×8359	5.90	12	6.32	6.26	**5.94**	6.54	6.10	5.95	5.95	6.14 (2.8)	6.08 (2.8)	**5.76** (2.9)	6.41 (2.0)	5.96 (2.3)	5.80 (2.4)	5.80 (2.4)
	Level 0 2†	IASI2	60×1528×8359	5.96	12	6.38	6.27	**5.96**	6.55	6.11	5.97	5.98	6.21 (2.7)	6.10 (2.8)	**5.79** (2.9)	6.43 (1.9)	5.98 (2.2)	5.83 (2.4)	5.85 (2.3)
	Level 0 3†	IASI3	60×1528×8359	5.25	12	6.31	6.19	**5.89**	6.48	6.04	5.90	5.91	6.14 (2.7)	6.01 (2.9)	**5.71** (3.0)	6.35 (2.0)	5.91 (2.2)	5.76 (2.3)	5.77 (2.3)
	Level 0 4†	IASI4	60×1528×8359	6.30	12	6.43	6.37	**6.04**	6.65	6.20	6.06	6.07	6.25 (2.9)	6.19 (2.8)	**5.87** (2.9)	6.52 (2.0)	6.07 (2.2)	5.92 (2.3)	5.91 (2.6)
	Average			5.85		6.36	6.27	**5.96**	6.56	6.11	5.97	5.98	6.19 (2.8)	6.09 (2.8)	**5.78** (2.9)	6.43 (2.0)	5.98 (2.2)	5.83 (2.4)	5.83 (2.4)

†: Uncalibrated (U). ‡: Calibrated (C). *: S9: Simple-9, S16: Simple-16, PFD: PForDelta.

a fixed interval, and at each interval, the partial sums of the number of integers are computed. With these strategies, random cell access can be done in constant time. Note that, however, it may incur some overhead and this should be taken into consideration when used in a real-time application.

IV. EXPERIMENTAL RESULTS

In this section, we present the results for storing hyperspectral data with a k^2-raster structure, incorporating the aforementioned improvements in padding storage and integer encoding strategies.

The hyperspectral scenes were captured by different sensors in real remote-sensing missions: Atmospheric Infrared Sounder (AIRS), AVIRIS, Compact Reconnaissance Imaging Spectrometer for Mars (CRISM), Hyperion, and IASI. These images are often used in the remote sensing data compression literature. All of them, save for IASI, are publicly available for download (http://cwe.ccsds.org/sls/docs/sls-dc/123.0-B-Info/TestData). The tested hyperspectral scenes are listed in Table I. Note that the storage size for hyperspectral scenes is measured in bits per pixel per band (bpppb).

We extended the algorithms presented in the paper by Ladra *et al.* [11], and our k^2-raster implementation was based on these new extended algorithms. The DACs' software was downloaded from the Universidade da Coruña's Database Laboratory website (http://lbd.udc.es/research/DACS/). The original implementations for Simple-9, Simple-16, and PForDelta can be found on the website for the Poly-IR-Toolkit (https://code.google.com/archive/p/poly-ir-toolkit/source/default/source). However, to incorporate the random access function, the code was also modified and extended.

Programs for the experiments were written in C and C++ and compiled by GNU g++ 6.3.0 20170516 with -O3 optimization. The testing computer had an Intel(R) Xeon(R) 4-core CPU E3-1230 V2 @ 3.30 GHz with 8192 KB of cache and 32 GB of RAM. The operating system was Debian GNU/Linux 9 with kernel Linux 4.9.0-8-amd64. The software code is available at http://gici.uab.cat/GiciWebPage/downloads.php.

A. Storage Size With and Without Padding

In this section, we show how storage sizes fare among the different encoders using padded and unpadded matrices. The results are presented in Table I. It can be seen that k^2-raster produces smaller storage sizes than gzip. The table also shows that the sizes for unpadded matrices are less than the padded matrices with up to 6% in savings. PForDelta using 128-integer blocks has the best results for the majority of padded and unpadded matrices, followed by DACs and Simple-16 encoders. Overall, the storage size has been reduced for most data except for Hyperion scenes using PForDelta, where PForDelta-128 codes for padded matrices produce similar results for PForDelta-256 codes for unpadded matrices. Both are almost equal, and the difference is not significant. If we examine Table I, it can be seen that the use of an unpadded matrix with PForDelta-128 compared with a padded matrix with DACs results in a reduction for storage for almost all the test files except for CRISM. We believe the experiments have shown us that using an unpadded matrix together with one of integer encoders, in particular PForDelta, can help improve the storage size.

TABLE II

COMPARISON OF RANDOM ACCESS TIME (μs) BETWEEN DIFFERENT INTEGER ENCODERS FOR PADDED AND UNPADDED MATRICES. THE BEST RESULTS ARE HIGHLIGHTED. THE ABBREVIATIONS FOR THE INTEGER ENCODERS ARE THE SAME AS IN TABLE I .

Scene Data	Padded Matrix (μs)							Unpadded Matrix (μs)						
	DACs	S9	S16	PFD 32	PFD 64	PFD 128	PFD 256	DACs	S9	S16	PFD 32	PFD 64	PFD 128	PFD 256
AG9	0.65	0.65	0.65	0.62	0.59	**0.56**	0.66	**0.58**	0.67	0.66	0.60	0.62	0.62	0.70
AG16	0.58	0.67	0.60	0.57	**0.56**	0.57	0.60	**0.55**	0.64	0.65	0.58	0.61	0.63	0.71
AG60	0.63	0.67	0.69	**0.56**	0.61	0.61	0.67	**0.55**	0.64	0.67	0.59	0.60	0.65	0.73
AG126	0.51	0.69	0.64	0.59	0.50	**0.49**	0.62	**0.60**	0.65	0.64	0.62	0.63	0.66	0.73
AG129	0.67	0.67	0.69	**0.57**	0.58	**0.57**	0.63	**0.58**	0.63	0.62	0.59	0.59	0.61	0.70
AG151	0.68	0.64	0.67	**0.54**	0.57	0.60	0.68	**0.60**	0.65	0.65	0.61	0.62	0.63	0.72
AG182	0.69	0.67	0.72	**0.55**	0.61	0.57	0.78	**0.67**	**0.67**	0.68	0.75	0.74	0.80	0.89
AG193	0.57	0.65	0.68	0.58	0.57	**0.56**	0.69	**0.55**	0.65	0.64	0.60	0.60	0.65	0.70
Average	0.62	0.66	0.67	0.57	0.57	**0.56**	0.67	**0.58**	0.65	0.65	0.62	0.63	0.66	0.73
AC00	1.59	1.63	1.63	**1.47**	1.51	1.67	1.67	0.71	0.77	0.77	**0.70**	0.71	0.73	0.79
AC03	1.54	1.58	1.62	1.45	**1.37**	1.47	1.58	**0.69**	0.77	0.77	0.71	0.71	0.73	0.79
AC10	1.91	1.63	1.65	1.62	1.59	**1.34**	1.51	0.89	0.76	0.76	**0.68**	**0.68**	0.69	0.76
AC11	1.83	1.64	1.63	**1.36**	1.52	1.49	1.48	0.86	0.77	0.77	**0.70**	0.71	0.71	0.76
AC18	1.59	1.62	1.64	**1.44**	1.53	1.45	1.59	0.72	0.78	0.77	**0.70**	**0.70**	0.73	0.78
Average	1.69	1.62	1.64	**1.47**	1.50	1.48	1.56	0.78	0.77	0.77	**0.70**	**0.70**	0.72	0.78
AU00	0.93	1.05	1.05	0.99	**0.91**	1.01	0.92	0.60	0.65	0.66	**0.59**	**0.59**	0.61	0.66
AU03	0.92	1.05	1.04	**0.84**	0.89	0.99	0.92	**0.59**	0.65	0.65	**0.59**	**0.59**	**0.59**	0.62
AU10	1.24	1.05	1.06	0.92	0.97	**0.87**	1.01	0.78	0.64	0.64	**0.57**	**0.57**	**0.57**	0.65
AU11	1.13	1.05	1.02	0.95	0.94	**0.87**	1.05	0.73	0.66	0.65	**0.57**	**0.57**	0.58	0.60
AU18	0.99	1.10	1.10	**0.88**	0.99	0.99	0.98	0.61	0.66	0.66	0.59	**0.58**	0.59	0.62
Average	1.04	1.06	1.05	**0.91**	0.94	0.94	0.98	0.66	0.65	0.65	**0.58**	**0.58**	0.59	0.63
CR1	1.66	1.64	1.62	**1.43**	1.52	1.56	1.60	0.82	0.88	0.89	**0.78**	**0.78**	0.80	0.87
CR2	1.71	1.60	1.74	1.50	**1.41**	1.65	1.51	0.85	0.89	0.90	0.81	**0.80**	0.81	0.89
CR3	1.59	1.72	1.73	1.51	**1.49**	1.52	1.53	**0.80**	0.92	0.91	0.81	0.81	0.81	0.87
CR4	1.43	1.49	1.42	1.22	**1.20**	1.26	1.37	0.82	0.81	0.81	0.74	**0.73**	0.75	0.83
CR5	1.48	1.51	1.54	**1.22**	1.26	1.27	1.42	0.84	0.82	0.81	**0.73**	0.74	0.75	0.84
CR6	1.44	1.50	1.53	1.32	**1.29**	1.32	1.44	0.77	0.82	0.82	**0.73**	**0.73**	0.76	0.83
Average	1.55	1.58	1.60	1.37	**1.36**	1.43	1.48	0.82	0.86	0.86	0.77	**0.76**	0.78	0.85
HC1	1.52	1.64	1.63	1.53	**1.44**	1.49	1.65	**0.62**	0.71	0.71	0.64	0.65	0.67	0.72
HC2	1.63	1.56	1.49	**1.47**	1.49	1.48	1.68	0.71	0.70	0.70	0.64	**0.63**	0.65	0.69
HC3	1.38	1.47	1.50	1.40	1.40	1.47	**1.36**	**0.63**	0.70	0.70	0.64	0.64	0.65	0.70
Average	1.51	1.56	1.54	1.47	**1.44**	1.48	1.56	0.65	0.70	0.70	**0.64**	**0.64**	0.65	0.70
HU1	1.91	2.02	2.02	1.70	1.72	**1.61**	1.76	**0.79**	0.88	0.89	**0.79**	**0.79**	0.80	0.84
HU2	1.82	2.01	2.00	1.76	**1.68**	1.82	1.85	**0.78**	0.89	0.88	0.79	0.79	0.81	0.87
HU3	1.82	2.02	2.05	1.82	1.83	**1.77**	1.87	**0.76**	0.89	0.89	0.79	0.78	0.80	0.85
Average	1.85	2.02	2.02	1.76	1.74	**1.73**	1.83	**0.78**	0.89	0.89	0.79	0.79	0.80	0.85
IASI1	1.34	1.24	1.30	1.10	**1.02**	1.05	1.09	**1.25**	1.46	1.32	1.27	1.31	1.38	1.44
IASI2	1.34	1.29	1.27	**1.12**	**1.12**	**1.12**	**1.12**	**1.25**	1.47	1.30	1.33	1.30	1.39	1.43
IASI3	1.35	1.31	1.29	1.11	1.12	**1.02**	1.13	**1.29**	1.42	1.30	1.31	1.31	1.40	1.45
IASI4	1.38	1.32	1.24	1.11	**1.06**	1.09	1.14	**1.27**	1.43	1.34	1.32	1.33	1.35	1.45
Average	1.35	1.29	1.28	1.11	1.08	**1.07**	1.12	**1.27**	1.45	1.32	1.31	1.31	1.38	1.44

B. Random Query With and Without Padding

In this section, we test the time it takes to query elements. Random access to elements in each tested scene is performed in 100 000 iterations with and without padding. The GetCell function in the paper by Ladra *et al.* [11] is modified using partial sums and sampling to optimize random cell access time in an unpadded matrix. To ensure more accurate results, we use the same set of coordinates and bands generated randomly for each scene for all the different encoders and matrices. The program is repeated 20 times for each scene and the average time is taken. The results are shown in Table II. It shows that the best access time is to use the 32-, 64-, and 128-integer PForDelta encoders for padded matrices and DACs, and 64- and 32-integer PForDelta encoders for unpadded matrices. What is more notable is that the access times are cut almost in half (49%) for most of the unpadded matrices, whereas AIRS Granules and IASI data have more or less the same access times for both types of matrices, possibly due to the similar tree data built from the padded and unpadded matrices. Other factors influencing execution times are the effectiveness of the partial sums and sampling optimization, and the relatively small spatial sizes of AIRS and IASI matrices (e.g., AIRS has $90 \times 135 = 12\,150$ pixels) compared with others (e.g., AVIRIS Yellowstone has $680 \times 512 = 348\,160$ pixels). Thus, the effects are less noticeable. Also, we should note that for access times in unpadded matrices, the way the tree is traversed is greatly boosted using the partial sums and sampling, which help access elements faster.

These experiments prove that the storage space and random access to elements in the k^2-raster structure produce competitive results not only for DACs but also for the Simple family of word-aligned integer encoders.

V. CONCLUSION

In this research, we propose a new no-padding method to reduce the storage space by saving only the elements in the nodes of a k^2-raster that are within the bounds of the original matrix, and our experiments have shown that it saves space up to 6%. The access time has also been reduced by half for most of the data when using unpadded matrices. Furthermore, the use of other random access integer encoders, such as Simple-9, Simple-16, and PForDelta, has proven to be competitive compared with DACs, the encoder originally used by the authors of k^2-raster. In particular, we can see that for most hyperspectral data, PForDelta performs better than DACs with up to 6% reduction in storage size and up to 20% reduction in random access time to elements. The experiments also show that the Simple family of integer encoders can also be used as a good alternative to DACs for random access to integer sequences.

REFERENCES

[1] G. Navarro, *Compact Data Structures: A Practical Approach*. Cambridge, U.K.: Cambridge Univ. Press, 2016.

[2] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. 30th Annu. Symp. Found. Comput. Sci.*, Oct. 1989, pp. 549–554.

[3] *Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression. Blue Book. Issue 2, Consultative Committee for Space Data Systems (CCSDS)*, Standard CCSDS 123.0-B-2, Feb. 2019. [Online]. Available: https://public.ccsds.org/Pubs/123x0b2c3.pdf

[4] K. Chow, D. Tzamarias, I. Blanes, and J. Serra-Sagristà, "Using predictive and differential methods with K^2-raster compact data structure for hyperspectral image lossless compression," *Remote Sens.*, vol. 11, no. 21, p. 2461, Oct. 2019.

[5] N. R. Brisaboa, S. Ladra, and G. Navarro, "DACs: Bringing direct access to variable-length codes," *Inf. Process. Manage.*, vol. 49, no. 1, pp. 392–404, Jan. 2013.

[6] N. R. Brisaboa, M. R. Luaces, G. Navarro, and D. Seco, "A fun application of compact data structures to indexing geographic data," in *Proc. Int. Conf. Fun With Algorithms*. Berlin, Germany: Springer, 2010, pp. 77–88.

[7] N. Brisaboa, A. Fariña, G. Navarro, and J. Paramá, "Dynamic lightweight text compression," *ACM Trans. Inf. Syst.*, vol. 28, no. 3, pp. 1–32, Jun. 2010.

[8] N. R. Brisaboa, S. Ladra, and G. Navarro, "Compact representation of Web graphs with extended functionality," *Inf. Syst.*, vol. 39, pp. 152–174, Jan. 2014.

[9] N. R. Brisaboa, M. A. Rodríguez, D. Seco, and R. A. Troncoso, "Rank-based strategies for cleaning inconsistent spatial databases," *Int. J. Geograph. Inf. Sci.*, vol. 29, no. 2, pp. 280–304, Feb. 2015.

[10] F. Silva-Coira, "Compact data structures for large and complex datasets," Ph.D. dissertation, Facultade de Informática Universidade da Coruña, A Coruña, Spain, 2017.

[11] S. Ladra, J. R. Paramá, and F. Silva-Coira, "Scalable and queryable compressed storage structure for raster data," *Inf. Syst.*, vol. 72, pp. 179–204, Dec. 2017.

[12] K. Chow, D. E. O. Tzamarias, M. Hernández-Cabronero, I. Blanes, and J. Serra-Sagristà, "Analysis of variable-length codes for integer encoding in hyperspectral data compression with the k^2-raster compact data structure," *Remote Sens.*, vol. 12, no. 12, p. 1983, Jun. 2020.

[13] V. N. Anh and A. Moffat, "Inverted index compression using word-aligned binary codes," *Inf. Retr.*, vol. 8, no. 1, pp. 151–166, Jan. 2005.

[14] J. Zhang, X. Long, and T. Suel, "Performance of compressed inverted list caching in search engines," in *Proc. 17th Int. Conf. World Wide Web (WWW)*, 2008, pp. 387–396.

[15] M. Zukowski, S. Heman, N. Nes, and P. Boncz, "Super-scalar RAM-CPU cache compression," in *Proc. 22nd Int. Conf. Data Eng. (ICDE)*, Apr. 2006, p. 59.