OGC® DOCUMENT: 24-041

External identifier of this OGC® document: http://www.opengis.net/doc/PER/t20-D011

Open Geospatial Consortium

OGC TESTBED-20 GIMI BENCHMARKING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2025-05-06
Approval Date: 2025-06-12
Publication Date: 2025-05-20
Editor: Sina Taghavikish

Note

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I KEYWORDS

The following are keywords to be used by search engines and document catalogues.

testbed, GIMI, benchmarking, COG, GeoTIFF, TIFF, HEIF, AVIF

II CONTRIBUTORS

All questions regarding this document should be directed to the editor or the contributors:

NAME	ORGANIZATION	ROLE
Sina Taghavikish	OGC	Editor
Martin Desruisseaux	Geomatys	Contributor
Brad Hards	Silvereye Technology	Contributor
Nicholas Kellerman	Helyx	Contributor
Carl Reed	OGC	Contributor

III OVERVIEW

This OGC Testbed 20 GIMI Benchmarking Report documents the evaluation of implementations of the OGC Geographic Tagged Image File Format (GeoTIFF) Standard and Cloud Optimized GeoTIFFs (COGs) to see how they perform compared to the new GEOINT Imagery Media for ISR (GIMI) standard. The Testbed 20 GIMI participants created various datasets using data from satellites such as Sentinel-1 and Landsat and for comparison synthetic data. Synthetic data are artificially generated rather than produced by real-world events.

COGs were tested under controlled conditions and practical situations. The participants noted challenges such as managing complex data during conversion and limitations of support in certain programming environments. Also, using implementations of the High Efficiency Image Format (HEIF) can significantly reduce file sizes compared to PNG tiles. This makes GeoHEIF comparable to PNG even for simple files when combined with a low-overhead image file format. Furthermore, the assessment included comparing the performance of reading GeoTIFF and

GeoHEIF files using a custom pure-Java reader and a C/C++ GDAL library binding in a Java environment. The experiment highlighted the need for repeated full benchmarks and the use of specialized tools for accurate statistical analysis due to variations in execution time. The focus was on understanding the costs and efficiency of these different methods, particularly in terms of how data is managed across programming languages.

IV FUTURE OUTLOOK

The Testbed 20 GIMI Task on GIMI and GeoTIFF benchmarking indicates that future developments in geospatial data formats and tools are essential to meet the needs of complex imagery products. A key priority is working on formats such as Cloud Optimized GeoHEIF (COHEIF), which offers greater flexibility for managing detailed metadata. Another priority is enhancing support for formats such as HEIF in programming languages such as Python to promote broader adoption. Additionally, further benchmarking is required when COHEIF is ready for opportunities to optimize interactions with native libraries, reducing overhead and enhancing efficiency in geospatial data processing. Focusing on these areas will advance the standardization and interoperability of geospatial imagery formats, leading to more effective and efficient data handling solutions for businesses. Overall, future efforts will focus on creating better tools to handle complex data, ultimately aiming to support broader use and accessibility in the geospatial field.

V VALUE PROPOSITION

The Testbed 20 GIMI benchmarking findings indicate that HEIF, combined with high-efficiency codecs and low-overhead formats, is a promising solution for modern raster image compression needs, providing both significant storage savings and quality retention. By analyzing real and simulated data, this work helps the geospatial community better understand the strengths and weaknesses of COG, GeoTIFF and GeoHEIF formats and suggests improvements for better handling of data. The Testbed 20 GIMI task supported the creation of new (draft) specification, ultimately improving how data is shared and accessed within the geospatial community. Overall, these efforts aimed to enhance efficiency and performance in managing large geospatial data sets, which can lead to significant benefits for organizations in this field.

1 INTRODUCTION

1 INTRODUCTION

Initially, a list of potential benchmarking scenarios was generated. The list could be categorized into three sections:

- Scenarios implemented;
- Scenarios not implemented in Testbed but are possible; and
- Scenarios that could not be done in Testbed 20.

The two first sections of this Report contain scenarios that can be automated, measured and compared between different formats and implementations. The last section contains, for example, scenarios about user feeling (the perceived performance), which are hard to measure. The perceived performance can be improved by progressive rendering, even if the real performance is the same or even slower.

A. Scenarios Implemented:

The following is a list of the benchmarking scenarios that were implemented and executed during the Testbed 20 GIMI Task. Note that some scenarios are considered equivalent and mapped to the same code[OGC 24-039, Appendix C].

1. **Serial Access:** In this scenario, sequential access to image data is the focus. For instances such as streaming or linear data processing, the requirement for precise conversion between user-requested coordinates and actual pixel positions can be somewhat relaxed. The system assesses performance under these circumstances, allowing for greater flexibility in managing HTTP range requests that retrieve sections of data without the necessity for pixel-perfect accuracy.

2. **Random Access:** This scenario emphasizes the capability to extract a Region of Interest (ROI) from a very large image dataset. Often, the ROI does not align perfectly with the image tile boundaries, which can impact the efficiency of data retrieval. The evaluation in this scenario focuses on how adeptly the system manages such conditions, measuring factors such as speed and resource utilization when accessing regions of data that are not aligned.

3. **Random Access with Overviews:** In this scenario the emphasis is on accessing various scale versions of the same image, referred to as overviews. The overview feature enables users to zoom in or out on an image and retrieve appropriately sized data segments. The system's performance is evaluated based on its responsiveness and efficiency in adapting to differing resolution requirements while ensuring a seamless user experience as they interact with varying scales of image data.

4. **Full Access:** In this scenario, the complete download of an entire image is examined, analyzing how effectively the system manages such significant data transfers. Additionally, this scenario encompasses the processes involved in uploading and ingesting all related data into the system for subsequent

processing or analysis. Performance metrics in this context focuses on throughput, latency, and overall system reliability when handling large volumes of data in single operations.

5. **Extraction of Metadata:** All the above scenarios include the extraction of metadata for the entire dataset. This allows users to access detailed information about the image and image tile information.

6. **Compressed and Uncompressed Data:** In addition to the above scenarios, both compressed and uncompressed data formats were considered.

7. **Synthetic vs Real Data:** Real data provides an honest representation of data values, which affects the variables associated with data compression formatting. For example, common occurrences such as NoData areas outside of satellite imaging swaths are captured as "real" data. Synthetic data, on the other hand, removes the uncertainty that arises from unpredictable data content and its effect on compression.

   NOTE: Please note that the definitions of synthetic and real data in this Report are specific to this document. Real data is derived from genuine events, sensors, or user interactions. It captures the disorder, variability, and intricacy of real-world systems, which makes it perfect for testing robustness and managing unforeseen challenges. This data is crucial for assessing how systems operate under realistic conditions.

   In contrast, synthetic data is generated artificially to replicate the structure or behavior of real data. It provides complete control and scalability, which is beneficial for testing edge cases, targeted scenarios, and file format compatibility.

8. **Comparing GIMI Against Widely Implemented Formats:** Running a selection of the scenarios to evaluate the performance of HEIF, GeoHEIF, and COHEIF against TIFF, GeoTIFF, and COG.

B. Scenarios not implemented in Testbed 20 but are possible:

1. **Parallel Access Optimization**: In this scenario, parallel access to various segments of the file is considered. This is especially the case for scenarios such as machine learning inference where quick data retrieval is crucial.

2. **Targeted Metadata Extraction**: The ability to extract metadata for a specific ROI would be benchmarked in this scenario.

3. **Clip and Ship**: The "clip and ship" method involves selecting and transferring certain parts of an image along with its associated metadata to be utilized on another system. This process includes converting the data into different formats, such as transcoding to GIMI or NITF (National Imagery Transmission Format) or COG.

C. Scenarios that could not be done in Testbed 20:

C.1. Related to image stream/video

1. **Accessing an Image Stream:** This scenario involves extracting a specific frame or clip from a live or recorded stream.

2. **Searching by Video Metadata:** Search by video metadata to extract frames/clips which match metadata values/ranges. By querying this metadata, users can identify and extract frames or clips that match specified values or ranges, making it easier to locate relevant visual segments based on metadata.

3. **H266 requirement:** Benchmarking the above scenarios based on the H266 High definition motion imagery compression standard.

   C.2. User's Feeling:

4. **Scroller:** Interactive and user-friendly way to view content dynamically by scrolling across the image.

   C.3. Require write operations:

5. **Update on each Extraction:** In all extraction scenarios, do an update for each extraction performed.

   This Report provides a detailed review of the results from the implemented scenarios. It also covers scenarios that have not been mentioned above, such as comparing the performance in GDAL and Apache SIS. Due to limited time and resources in Testbed 20, only a fraction of the identified benchmarks were completed. For example, all the video-related scenarios were not feasible since the code for image sequences was under development while benchmarking was being conducted.

1.1. Aims

To benchmark GIMI/GeoHEIF against well-established formats such as TIFF, GeoTIFF, and COG.

1.2. Objectives

- Assess the performance of high-efficiency codecs (e.g., HEIF and AVIF) in comparison to traditional formats like PNG, focusing on storage and bandwidth efficiency for various image types and tile sizes.
- Conduct benchmarking tests on different geospatial formats, including Cloud-Optimized GeoTIFFs (COGs), TIFF, and HEIF.

- Perform benchmarking using both real and synthetic geospatial data.
- Compare the performance of GeoTIFF and GeoHEIF files.

NOTE:High-efficiency codecs, like HEVC (H.265) and AV1, are video compression standards that significantly improve compression efficiency compared to older codecs like H.264 (AVC). They achieve this by using more advanced algorithms that reduce file sizes while maintaining high video quality, especially important for streaming and storing high-resolution content like 4K and 8K (Mixilab 2025).

2

TOPICS

2 TOPICS

2.1. Silvereye Technology — Benchmarking with HEIF and High-Efficiency Codecs

Silvereye Technology's contribution to Testbed 20 benchmarking contribution highlights significant advancements in image file compression efficiency. Their contributions included the provision of test data and evaluations of small tiles, focusing on the advantages of high-efficiency codecs combined with a proposed low-overhead MPEG image file format. This combination resulted in a substantial reduction in file size compared to traditional formats such as PNG, demonstrating the potential for improved storage and bandwidth efficiency.

2.1.1. Test Data Contributions

Silvereye Technology converted GeoTIFF source data to HEIF format, enriched with GeoHEIF metadata. This dataset comprised high-resolution true-color aerial imagery of a residential area in Canberra, Australia. By employing custom tools to produce files with varying encoding options, Silvereye benchmarked assessing these options under diverse scenarios. This meticulous preparation ensured that evaluation results were attributable to encoding strategies rather than data variations.

2.1.2. Evaluation of Small Tiles

The focus on small tiles, such as those used in raster mapping applications like OpenStreetMap, revealed the capability of HEIF to outperform PNG in terms of file size. Two test tiles—one simple (a single-color forested area) and one complex (a commercial area with roads, buildings, and text)—were compressed using the AV1 codec with both standard and experimental low-overhead options. The low-overhead option notably reduced file sizes, making AVIF a competitive alternative to PNG, particularly for simple content.

2.1.3. Compression Results

The results highlighted that while PNG's lossless compression offers consistent quality, HEIF/AVIF provided significant file size reductions. For simple tiles, AVIF achieved parity with PNG, and for complex tiles, it produced smaller files at high quality levels. Lower-quality settings resulted in visual artifacts, but moderate quality settings were visually comparable to PNG while offering up to 50% smaller file sizes.

2.2. HELYX — BENCHMARKING COG vs TIFF and HEIF

Helyx benchmarked as extensively as possible the performance of Cloud-Optimized GeoTIFFs (COGs), particularly in comparison to TIFF and HEIF. Benchmarking was conducted against various COG datasets and creation options and, where feasible, against standard TIFF and HEIF files. Because the GDAL-powered COG drivers are tailored for different applications than the existing standard TIF and HEIF drivers, there were limited comparable scenarios. For instance, libheif is designed to support qualitative applications (human visualization), featuring specific bit depths and band counts, as well as supporting lossy compression. In contrast, COG and GeoTIFF are intended for quantitative applications (scientific analysis), offering a wide range of bit depths, the capacity for multiple bands, and supporting lossless compression.

2.2.1. Benchmarking Data

Helyx utilized two types of COG data for benchmarking: real data and synthetic data. Real data is beneficial for Real Data Benchmarking because it offers an accurate depiction of data values, impacting the variables related to data compression formatting. It effectively captures common occurrences like NoData areas outside swaths. On the other hand, synthetic data serves a purpose in Synthetic Data Benchmarking by alleviating the uncertainty caused by unpredictable data content and its effects compression.

2.2.2. Benchmarking Objectives

1. Synthetic Data Benchmarking assesses the direct performance of the COG and HEIF drivers. These tests strive to minimize external influences on test metrics, including network latency, variations in data content (which can impact compression), and complexities in driver implementation (for instance, libtiff used by GDAL when called by RasterIO in Python compared to libtiff utilized by PIL).

2. Real Data Benchmarking evaluates the performance of the COG and HEIF drivers under real-world conditions. These benchmarks aim to validate the Synthetic Data Benchmark tests by either confirming their accuracy or uncovering unknown factors that may impact performance, necessitating further investigation.

2.2.3. Benchmarking Scenarios

Each benchmark scenario performed by Helyx altered either a format variable, the image content, or an operational variable. Format variables pertain to the formatting of files and their metadata, while image content concerns the data type and pattern of raster data, both of which impact compression. Operational variables describe how the files are interacted with. Scenarios result from a blend of potential operations and formatting choices. Possible operations stem from basic read/write actions, including COG-specific operations like subset reading. Possible formatting choices come from the GDAL COG driver options.

Helyx utilized Jupyter Notebooks for benchmarking, employing two primary approaches: synthetic and real benchmarking. Synthetic COG and HEIF Benchmarking performed Synthetic Data Benchmarking using synthetic COGs on a local Docker network, while Real COG Benchmarking evaluated publicly accessible COGs from the Microsoft Planetary Computer for Read Data Benchmarking. The notebooks produce metrics and graphs to assist in interpreting results. Real tests measure read/write times against predefined data files without control over their content, whereas synthetic tests allow for controlled variables, including data creation patterns (such as random Gaussian noise or Mandelbrot fractals) and file formats (Rasterio/GDAL COG or PIL TIF/HEIF). Additionally, some benchmarking scenarios specify array dimensions, data types, compression algorithms, overview levels, and block sizes to configure benchmarking tests.

2.2.4. Benchmarking Results

- **Synthetic Data: Formats and Drivers**

The PIL TIFF format is the fastest for both reading and writing, although it is less efficient in terms of file size, potentially because of improper compression settings. In contrast, the RasterIO COG exhibits slower write times (0.1s compared to 0.02s for PIL) and significantly slower read times (0.046s compared to 0.005s). Meanwhile, the PIL HEIF format, while slower to write than TIFF due to HEVC compression, proves to be the most efficient regarding file size. Overall, a balanced comparison between these formats should take storage efficiency into account, and for applications where speed is critical, simpler compression techniques may be more suitable.

- **Synthetic Data: Image Dimensions**

The findings indicate that smaller images have significantly faster read and write times than larger images. While HEIF typically provides the most effective file size efficiency, there are instances where RasterIO's TIFF with JPEG compression outperforms HEIF. To enhance understanding of performance and storage, conducting multi-threaded benchmarks for larger images along with evaluations of lossy compression is recommended.

- **Synthetic Data: Bit Depths**

The findings indicate that read/write times for COG are proportional to bit depth, with longer durations observed at higher bit depths. HEIF performance is only comparable for 8-bit images, where it is slower than COG while producing smaller files. However, the advantages of HEIF in terms of file size diminish when dealing with high bit depths and lossless compression, rendering it less suitable for quantitative image analysis. Please note that this might be due to the PIL implementation.

- **Synthetic Data: Bit Depth and Compression**

The findings indicate that HEIF read/write times and file sizes are stable regardless of bit depth, while COG performance is influenced by bit depth and compression type. Among different methods, no compression provides the quickest write times, but JPEG compression results in

faster read times. Conducting further benchmarking on lossy compression methods, with a particular focus on HEVC, is recommended.

- **Real Data**

This Report analyzed read/write times and file sizes when retrieving STAC (Spatio-Temporal Asset Catalog) items over the internet, referring to the process of accessing these STAC-described data assets via the Planetary Computer's API. STAC is an open specification that standardizes how metadata for spatio-temporal assets—such as satellite imagery, drone data, or environmental datasets—is structured, catalogued, and queried, enabling consistent discovery and access across various platforms. A STAC server implementation provides a RESTful API that supports query operations and returns a JSON-based catalog structure to a client. STAC supports several domain-specific extensions, such as the Earth Observation (EO) extension, which add extra search fields like cloud cover, bands, and ground sampling distance (GSD). In this Report, STAC was utilized for cataloguing and retrieving spatio-temporal EO data.

The Microsoft Planetary Computer stores its extensive datasets in Azure and exposes them through a STAC-compliant API, allowing users to search, filter, and access data by location, time, general, and domain-specific metadata in a standardized manner.

The findings indicate that there is no clear relationship between read/write time or file size when retrieving STAC items over the internet. Therefore, it is recommended that more thorough benchmarking is conducted over extended periods to account for variations in internet traffic, and that comparisons of geo-enabled HEIF would be more meaningful than isolated assessments of COG.

2.3. Geomatys — GeoTIFF versus GeoHEIF

Geomatys conducted a benchmarking experiment to assess the performance of reading GeoTIFF and GeoHEIF files in a Java environment. Two approaches were tested:

- A pure-Java reader developed specifically for Testbed 20,
- A binding to the C/C++ GDAL library.

Benchmarks were performed comparing GeoTIFF versus GeoHEIF. The two libraries (pure-Java versus GDAL) were treated separately because interactions between the two languages (Java and C/C++), such as the need to use a binding framework (Annex B.5) and constraints during the transfers of raster data (Annex B.1.2), have an impact which is difficult to separate from intrinsic performances. These interactions are partially discussed in this Report.

Benchmark results are presented mostly for the Java implementations for two reasons: First is the fact that the Java implementations (Apache SIS) of the GeoHEIF and GeoTIFF readers share a considerable common code. This is *assumed* to reduce the risk that differences in performance are due to differences in implementation (this assumption is difficult to verify). Second, Apache SIS has a different strategy than GDAL regarding data reorganizations. At the cost of serving data that may not be exactly what the user requested, Apache SIS is more conservative about

data transformations (Clause 2.3.2). This policy reduces the risk that a difference in performance was caused by a request accidentally launching an operation more costly than intended.

2.3.1. Native Function Invocation Costs in Java

This Report discusses the costs associated with invoking native C/C++ functions from Java, particularly in light of the transition from the Java Native Interface (JNI) to Panama technology ('java.lang.foreign' package). Panama, introduced in Java 22, aims to simplify interfacing with the C language, but some performance overhead is still hardly avoidable. In particular, interactions between Java and C implies that copies of potentially large arrays of raster data for isolating native resources from a garbage collector activity. Panama gives explicit control on those copy operations, including the possibility to avoid copies in some circumstances. Therefore, the benchmarks used Panama for the GDAL binding rather than the OSGeo GDAL-Java binding. The latter is based on SWIG (Simplified Wrapper and Interface Generator) which itself uses JNI. This Report pays particular attention to copy operations because the copy operation was observed as being significant when comparing the two versions of the pure-Java GeoHEIF reader.

2.3.2. Challenges in Data Transfers and Band Reorganization

In addition to copy operations required by cross-language communication, copy operations depend also on the API of the library and whether this API is accessible from Java. For user convenience, an API may choose to reorganize the data in two ways:

- By returning data for exactly the spatiotemporal extent and resolution requested by the user,
- By storing each band in a separated array, hiding the complexity of various layouts in the raster file.

One way to avoid restructuring raster data is to access pixel *values* (as opposed to the whole raster) through an abstraction layer. This is done by Java2D's API, which supports various data layouts, such as banded, pixel interleaved, or multi-pixel packed models. For a Java application, this abstraction layer is more difficult to establish through a binding to a native library compared to a pure Java implementation.

2.3.3. Gaps in Cost Estimation and Methodology

The benchmark has the following limitations:

- Executed on only a small set of images and scenarios.
- The GeoTIFF and GeoHEIF images use tiles of the same size, but it was not verified if these tiles were written in the same order. Tile order may impact the number of seeks during read operations.

- Pyramided images were not tested. Pyramids are supported by the GeoTIFF reader, but that reader was not mature enough in the GeoHEIF reader Java implementation.
- While all readers support some compression schemes, only an uncompressed image was tested. Note, however, that compressions were expected to have the same cost in both formats.
- While all readers support various pixel organizations, only the *Pixel Interleaved Sample Model* (in Java2D terminology) was tested.
- The cost of the binding between Java and C languages was not estimated due to the difficulty of defining an identity operation for calibration purposes.
- The full benchmark of 100 iterations over 10,000 queries has been repeated only a few times, and the results were discarded or combined manually.
- No specialized tools such as Java Microbenchmark Harness (JMH) has been used.

A recommendation for future work is to repeat the full benchmark at least 40 times, with a relaunch of the Java Virtual Machine ever time. This is needed for more reliable statistics because of the observed variations in execution times.

3 OUTLOOK

3 OUTLOOK

The benchmark study identified areas for further exploration, such as benchmarking alternative codecs (e.g., JPEG-2000, HEVC or EVC), assessing larger tile sizes (e.g., 512 pixels or 1024 pixels per side), and evaluating other raster content types (e.g., high bit depth single channel, or true color RGB). Additionally, factors like compression latency and energy efficiency could be studied to enhance practical applications, while providing an objective evaluation of the result using metrics such as PSNR and structural similarity.

Investigating multi-threaded read/write times for larger images would be useful to reflect real-world scenarios. This will probably have an impact on the time it takes to perform more complex compression/decompression. Investigating the effect of lossy compression would be beneficial as this is critical to qualitative image processing workflows (rendering basemaps, etc.). It is less relevant for scientific images where lossless compression is necessary.

In bit-depth compression of synthetic data, benchmarking did not consider lossless compression, where the HEVC algorithm is likely to excel. Therefore, running additional benchmarks on lossy compression would be another venue for future work. Further benchmarking of lossy compression algorithms would be worthwhile to more thoroughly understand the benefits of HEIF/HEVC.

Comparing a geo- and cloud-enabled HEIF format would be more meaningful than an isolated assessment of COGs. However, such a comparison should still be performed in a rigorous manner (e.g., over a longer time period) to isolate time-variant factors (like internet traffic). Also, a more rigorous investigation into real data on an open network is needed.

The Testbed 20 benchmarking has limitations due to being executed on a small set of images and scenarios. The order of tiles in images was not confirmed, which may affect read operations. Pyramided images were not tested, and only one sample model has been assessed. The cost of integrating Java and C was not estimated. The complete benchmark needs to be repeated more times for reliable statistics.

4

SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS

4 SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS

A thorough review was conducted to identify any potential security, privacy, and ethical concerns. After careful evaluation, it was determined that none of these considerations were relevant to the scope and nature of this report. Therefore, no specific measures or actions were required in these areas.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ISO/IEC: ISO/IEC 23008-12:2022, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 12: Image File Format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2022). https://www.iso.org/standard/83650.html.

[2] OGC Testbed 20 Coverage Format Selection Report, 2025.

[3] OGC Testbed 20 GIMI Open Source Report

A

ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

A ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

ACT	Australian Capital Territory
AOM	Alliance for Open Media
Apache SIS	Apache Spatial Information System
API	Application Programming Interface
AV1	Alliance for Open Media Video 1
AVIF	AV1 Image File Format
CC-BY	Creative Commons Attribution
CEOS	Committee on Earth Observation Satellites
COG	Cloud Optimized GeoTIFF
COHEIF	Cloud Optimized GeoHEIF
EO-1	Earth Observing 1
ESA	European Space Agency
EVC	Essential Video Coding
GDAL	Geospatial Data Abstraction Library
GEOINT	Geospatial Intelligence
GeoTIFF	Geographic Tagged Image File Format
GIMI	GEOINT Motion Imagery for ISR
GRD	Ground Range Detected
HEIC	High Efficiency Image Container
HEIF	High Efficiency Image File

IEC	International Electrotechnical Commission
ISO	International Organization for Standardization JNI: Java Native Interface
JPEG	Joint Photographic Experts Group
LZW	Lempel-Ziv-Welch
NASA	National Aeronautics and Space Administration
NITF	National Imagery Transmission Format
NRT	Near-Real-Time
OGC	Open Geospatial Consortium
OSGeo	Open Source Geospatial Foundation
OSM	OpenStreetMap
PIL	Python Imaging Library
PNG	Portable Network Graphics
PSNR	Peak Signal-to-Noise Ratio
RGB	Red Green Blue
ROI	region of interest
SAFE	Standard Archive Format for Europe
SAR	Synthetic Aperture Radar
STAC	Spatio-Temporal Asset Catalogs
SWIG	Simplified Wrapper and Interface Generator
USGS	United States Geological Survey
VH	Vertical-Transmit, Horizontal-Receive
VV	Vertical-Transmit, Vertical-Receive
ZSTD	Zstandard

B

ANNEX B (INFORMATIVE) GEOMATYS – BENCHMARKING

B

ANNEX B (INFORMATIVE) GEOMATYS — BENCHMARKING

Geomatys benchmarked the reading of GeoTIFF and GeoHEIF files in a Java environment using two different methods: A pure-Java reader developed for Testbed 20 and a binding to the C/C++ GDAL native library. The GIMI Lessons Learned and Best Practices Report [OGC 24-042] provides more details about these implementations.

The purpose of the benchmark was to compare the GeoHEIF format with the GeoTIFF format. However, any measured differences are mixes of differences due to the formats and differences due to the libraries used for reading each format. For example, using GDAL to read raster data indirectly used two libraries which were developed independently: 'libtiff' and 'libheif'. In an attempt to reduce (but not eliminate) the possible variations due to differences in implementations, benchmarking was first run with Apache SIS (Spatial Information System). This project developed pure Java GeoTIFF and GeoHEIF readers, with an effort to share, as practical, a common code base. Therefore, the Apache SIS readers are assumed to be less subject to differences in implementations. This assumption is difficult to verify and may be counter-balanced by the fact that the performance of Java applications tends to vary between runs more easily than the performance of C/C++ applications. This is because the optimizations performed by the Just In Time (JIT) compiler can vary under subtle conditions difficult to control. Note also that this is not an assumption about the quality of the libraries: an implementation may be uniformly good or bad, the goal here is that it is approximately uniform.

B.1. Informal observation on the cost of copies

A GeoHEIF reader was developed for Testbed 20 and (at the time of writing this Report) is available in the "incubator" group of modules of the Apache SIS code repository. The initial prototype, committed October 2024, shared only partially the code that could be shared with the existing GeoTIFF reader. Initial benchmarks (not reported in this Report) showed that this GeoHEIF prototype was approximately two times as slow as GeoTIFF for reading the same image. This difference was assumed to be caused by the GeoHEIF prototype performing more copies and data reorganizations from one array to another. A major refactoring of the prototype, committed in February 26th, has brought the GeoHEIF reader to the same level as the GeoTIFF reader in terms of efforts for reducing array copies. Benchmarking on a local file then showed differences within the standard deviations. This observation explains the emphasis regarding array copy operations documented in this Annex.

B.1.1. Raster data reorganization

Raster data are often copied for two reasons:

- Merging tiles in one continuous array; and
- Using a separate array for each band.

This layout is common in scientific applications, since storing each band (or "variable" in netCDF) in one continuous array allows efficient iterations over the values. An alternative to data reorganization is to not read the values by direct access in an array, but instead through an API providing an abstraction layer for accessing data as they are stored. The Java2D standard API has a concept of "sample model" which makes the library quite flexible regarding data layout.

- **Banded Sample Model** for images where each band is stored in a separate array.
- **Pixel Interleaved Sample Model** for images where, for each pixel, the values of all bands are consecutive. Example: Red, Green and Blue of pixel 1, followed by Red, Green and Blue of pixel 2, *etc.*
- **Multi-Pixel Packed Sample Model** for images where many pixels are packed in a single number. For example, a bilevel mask can store 8 pixels per byte using one bit per pixel.

Java applications can use this flexibility for (as much as possible) storing pixel values in memory with the same layout as they are read from the file, without reorganizing the values. For example, when reading an uncompressed GeoTIFF image in the "chunky" format, Apache SIS directly uses the data provided by operating system I/O operation. This is done without reorganizing these data as separated bands and without merging the tiles in a continuous array.

Note that the time saved at read time can be lost later. This is because iterations over pixel values become more costly: They have to support different pixel layouts (interleaving models) and handle the split in many tiles. This complexity is hidden behind abstraction layers such as 'PixelIterator', but their use is more expansive than a plain iteration over an array. The net result depends on whether the application needs to access all pixels or only a subset, whether the application wants to use the existing split of data in tiles as a natural work unit for parallelization, and whether an algorithm implements shortcuts when its detects that the data layout is well suited, *etc.*

B.1.2. Cross-language transfers of arrays

The interfacing between Java and a native library has some performance costs. A part of the cost come from an additional copy of arrays. Copies may come from the mechanism used for crossing the language barrier (e.g., Java Native Interface) or may come from interactions between this mechanism and the API of the native library. When a client application and a library need to transfer large arrays such as large raster image, there are at least three possible strategies.

1. When invoking a library function, the client application provides a pointer telling the library where to read or write the data. This may be a pointer to the application's internal buffers. This approach requires that the application trusts the library.

2. Conversely, the library may return a pointer to its internal buffer, in which case the client application reads or copies the data itself. This approach requires that the library trusts the application.

3. Alternatively, the library may copy data to a temporary buffer, in which case the application reads or copies data from that buffer. With this approach, neither the library nor the application needs to trust the other.

The figure below illustrates these three strategies. Each arrow represents an array copy operation. The direction is from the software component that performs the read or write operation, toward the software component which exposes its internal for allowing that operation.

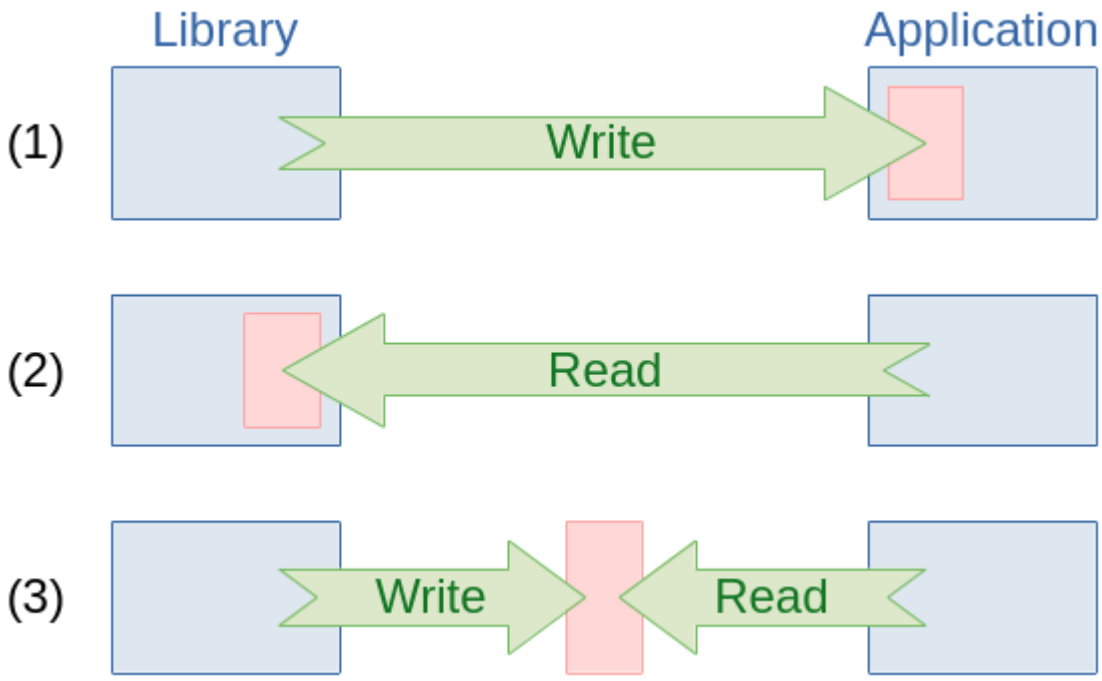

Figure B.1 — Strategies for transferring data from a library to a client applications

The GDAL native library generally applies the first strategy in its C/C++ API. The second strategy is also available through the `GetLockedBlockRef` function, which provides direct access to the GDAL internal cache. However, the latter method is available only through the C++ API, while the binding framework used in OGC Testbed 20 can use only the C API.

Since the Java Virtual Machine (JVM) does not give pointers to Java arrays, the first strategy is not possible (except in some special cases called "critical") with a binding though 'java.lang.foreign' . Said otherwise, the JVM generally does not allow a C/C++ library to write directly in a Java array, except in exceptional circumstances called "critical functions". One

reason is because the garbage collector may move the array to a new memory location at any time, and C/C+&#43 libraries are not prepared to handle such memory address changes in the middle of an operation. Therefore, the SIS-GDAL binding must allocate a temporary buffer, ask GDAL to copy some raster data to that buffer, then let Java copy the buffer's content to the final Java array. This is the third strategy in list above. The extra copy could be avoided with the second strategy, but the latter is not accessible in the SIS-GDAL binding case.

In summary, extra copy operations (compared to pure Java or pure C/C+&#43 implementations) were necessary in Testbed 20 when using a native library because of the interaction between the native API and the framework used for accessing it. In Java context, extra copies could be avoided if the native library allowed the client application to perform copies itself (strategy 2) through a C API.

B.2. Benchmarking method

The benchmarks described in this Annex were implemented by a Java application available as a Maven project on the Testbed code repository. The application read a text file which describes thousands of queries following a scenario. The scenarios implemented in the benchmarking code were:

- Reading tiles of 512 × 512 pixels sequentially (from left to right, then top to bottom) at full resolution;
- Reading tiles of 512 × 512 pixels at random positions but full resolution;
- Reading tiles of 512 × 512 pixels at random positions and random resolutions;
- Reading tiles of random sizes at random positions and resolutions but trying to keep the number of pixels per tile approximately constant;
- Reading tiles of random sizes at random positions and resolutions; and
- Reading the full image.

Each scenario can be generated once and saved in a text file for reproducibility. A different text file must be generated for each image to test, because the queries depend on the geospatial extent of the image. Each image should be available in two formats, GeoTIFF and GeoHEIF. The tester should verify that the following characteristics are identical in the GeoTIFF and GeoHEIF images (this was not verified by the benchmarking application).

- The size (in pixels) of the two images.
- The size (in pixels) of tiles in each image.
- The order in which the tiles are written.
- The color model (e.g., RGB, CMYB, palette or gray scale).

- The sample model (number of bands and interleaving mode).
- The data type of sample values (e.g., bytes, short integers or floating points).
- The compression algorithm (including the predictor if any).
- The Coordinate Reference System (CRS).
- The conversion from pixel to CRS coordinates.
- The number of overview levels if the image is pyramided.
- For each overview level, all above cited characteristics.

Due to lack of time, the benchmark described in this annex used a single image with the following characteristics.

- Image size of 9728 × 9216 pixels.
- Tile size of 512 × 512 pixels.
- RGB color model.
- Pixel interleaved sample model.
- Sample values stored as bytes.
- No compression.
- Transverse Mercator projection.
- No pyramid.

The Testbed 20 participants did not verify that the two images had the tiles written in the same order. This verification should be added in the future if this benchmarking effort is continued. An overview of the image is below (location is in Australia):

Figure B.2 — Test image

The following snippet shows the header of a scenario file followed by the 4 first queries. Each scenario file contains 10,000 such queries. The scenario shown below is the one for queries of random sizes, positions and resolutions.

```
# OGC Testbed 20 – GIMI and COG benchmark
```

```
# https://gitlab.ogc.org/ogc/T20-GIMI/
#
# GridGeometry
#   ├─Grid extent
#   │   ├─Column: [0 … 9727] (9728 cells)
#   │   └─Row:    [0 … 9215] (9216 cells)
#   ├─Geographic extent
#   │   ├─Lower bound:  35°19′23″S  149°06′03″E
#   │   └─Upper bound:  35°18′54″S  149°06′44″E
#   ├─Envelope
#   │   ├─Easting:   690,999.950000002 … 691,972.750000002   ΔE = 0.1 m
#   │   └─Northing: 6,089,078.45000004 … 6,090,000.05000004  ΔN = 0.1 m
#   ├─Coordinate reference system
#   │   └─EPSG:28355 — GDA94 / MGA zone 55
#   └─Conversion (origin in a cell center)
#       └─┌                                  ┐
#         │ 0.1   0     691000.000000002 │
#         │ 0    -0.1  6090000.000000040 │
#         │ 0     0          1           │
#         └                                  ┘
#
# Scenario: RANDOM

{
    areaOfInterest = 691263.650000002 6089473.6500000395, 691678.3500000021
6089681.6500000395
    resolution     = 1.3 1.3
    gridExtent     = 2626 3172, 6760 5239
    subsampling    = 13 13
    subsampledSize = 319 160
}

{
    areaOfInterest = 691652.950000002 6089078.45000004, 691948.950000002
6089826.45000004
    resolution     = 1.6 1.6
    gridExtent     = 6528 1728, 9472 9200
    subsampling    = 16 16
    subsampledSize = 185 468
}

{
    areaOfInterest = 691155.550000002 6089121.55000004, 691891.750000002
6089575.15000004
    resolution     = 1.8 1.8
    gridExtent     = 1548 4248, 8892 8766
    subsampling    = 18 18
    subsampledSize = 409 252
}

{
    areaOfInterest = 691304.450000002 6089272.45000004, 691625.050000002
6089920.6500000395
    resolution     = 1.4 1.4
    gridExtent     = 3038 784, 6230 7252
    subsampling    = 14 14
    subsampledSize = 229 463
}
```

Listing B.1 — Snippet of scenario file for the random case

While all scenarios listed at the beginning of this section were implemented and were available in the benchmarking application, only the two first scenarios were executed in Testbed 20: Reading tiles of 512 × 512 pixels sequentially, then at random positions but always at full resolution. Each scenario of 10,000 queries was executed about 100 times. The machine running the benchmark application had the following characteristics:

- Processors: Intel Core i7-8750H × 12
- Memory: 32 Gb
- Operation system: Fedora 41
- Linux 6.13.9-200.fc41.x86_64
- Apache HTTP server 2.4.63
- OpenJDK Runtime Environment: Red_Hat-24.0.0.0.36-1

The Apache SIS version used was a snapshot of the main development branch of version 1.5, built from commit 3950fbe8d5fc2dfac23a4b4a7bd0eb1d1ddea90b (April 11, 2025).

B.3. Running the benchmark

The benchmark application expects a specific directory structure. As a general rule, each directory contains data in one specific geographic area at one specific resolution, at least approximately. Then, an arbitrary number of sub-directories encode the same data in different formats, using different compressions, different tile sizes, *etc*. While different sub-directories were planned for testing different combinations of image layouts and compressions, only the following directory structure has been used in Testbed 20, with the 'ACT2017*' images downloaded and renamed from here.

```
<data directory>
└── COG versus HEIF
    └── No pyramid
        ├── ACT2017.heif
        └── ACT2017.tiff
```

Listing B.2 — Directory structure of test data

For building the benchmarking application, clone the Testbed 20 repository and execute the following commands (replace `cd` by `chdir` on Windows):

```
cd D011/code
mvn package
```

Listing B.3 — Build benchmarking application (Unix)

For generating the scenario files, execute the following commands (replace `<data directory>` by the root of above directory tree):

```
./run.sh generate-scenarios \
```

```
    --image-directory <data directory> \
    --images COMPARE_COG,COMPARE_HEIF \
    --scenarios SERIAL
```

Listing B.4 — Generate scenario files (Unix)

For running the scenario files, execute the following commands:

```
./run.sh benchmark \
    --image-directory <data directory> \
    --images COMPARE_COG,COMPARE_HEIF \
    --scenarios SERIAL \
    --num-passes 110 \
    --ignored-passes 10
```

Listing B.5 — Run benchmarking application (Unix)

Replace `SERIAL` by `RANDOM_WITH_FULL_RESOLUTION` for testing the scenario reading tiles at random positions. Add `--use-gdal true` for testing with GDAL instead of Apache SIS (it requires a GDAL installation to be reachable on the library path of the host computer).

B.3.1. Running over HTTP

The benchmarks were also executed with images read by HTTP range requests instead of local file access. This is done by starting an Apache HTTP server on the local host. On Linux systems, this can be done as defined below:

```
sudo systemctl start httpd.service
```

Listing B.6 — Start Apache HTTP server (Linux)

The directory structure containing the `ACT2017` images need to be copied to the `/var/www/html/` directory. Then, the benchmarks can be run again with the following command:

```
./run.sh benchmark \
    --scenario-directory <data directory> \
    --image-directory "http://localhost/" \
    --images COMPARE_COG,COMPARE_HEIF \
    --scenarios SERIAL \
    --num-passes 110 \
    --ignored-passes 10
```

Listing B.7 — Run benchmarking application over HTTP (Unix)

B.3.2. Consistency check

For every test, by computing the sum of pixel values it is possible to check that the different readers and scenarios have read the same thing. This check is enabled by adding the '—iterate-over-pixels -1' argument. The reported sums should be the same for GeoTIFF and GeoHEIF formats, and when using Apache SIS or GDAL libraries. An ordinary sum is used instead of MD5 sum in order to be insensitive to tile order and iteration order. For this Testbed, it has been verified that the results were consistent for the three bands in all tested cases.

Implementation note: The iteration over pixel values is known to be very slow in this benchmark application. The process could have been optimized, but it was not a benchmark goal. The

program allowed a compromise by specifying a positive argument value. For example, '—iterate-over-pixels 15' will iterate over only the first 15 pixels of each request.

B.4. Benchmarking results

The following table shows the execution times with the GeoTIFF and GeoHEIF pure-Java readers. 10,000 reads of tiles of 512 × 512 pixels were done, then the whole process was repeated about 100 times. The statistics of the first 10 iterations were discarded as the Java Virtual Machine warmup (this was conservative, as one iteration is usually sufficient). The scenarios shown in the table are:

- Sequential scenario (tiles read from left to right, then top to bottom);
- Sequential scenario again (for checking results stability); and
- Same tile sizes and same resolution, but at random positions.

Table B.1 — Execution times (milliseconds) of queries of 512 × 512 pixels on the local file system

	GEOTIFF (SEQ. 1)	GEOHEIF (SEQ. 1)	GEOTIFF (SEQ. 2)	GEOHEIF (SEQ. 2)	GEOTIFF (RANDOM)	GEOHEIF (RANDOM)
Minimum:	0.0030	0.0029	0.0030	0.0029	0.0033	0.0033
Maximum:	1.4683	2.1004	1.7757	1.6104	2.7342	2.6573
Average:	0.0048	0.0047	0.0046	0.0046	0.0214	0.0326
Std. dev:	0.0176	0.0209	0.0166	0.0210	0.0883	0.1119

The above table shows large variations, with relatively large maximum values and with standard deviations that are much larger than the average values. Note that even with the sequential scenario reading tiles from left to right then top to bottom, there is no guarantee that the tiles in the files are organized in that order. It is possible that the reading of some tiles where slower because they required a seek to a distant position in the file. This hypothesis has not been verified. For avoiding this uncertainty, the following table shows the statistics of execution times of iterations over all queries instead of statistics over the reading time of individual query. Since the exact same scenario was repeated for each iteration, there were no variation between each iteration, contrary to the time measurements of individual queries. The times shown in the table are the average times per query for making the comparison with the above table easier.

Table B.2 — Execution times (milliseconds) of image read operations on the local file system

	GEOTIFF (SEQUENTIAL)	GEOHEIF (SEQUENTIAL)	GEOTIFF (RANDOM)	GEOHEIF (RANDOM)
Minimum:	0.00429	0.00418	0.02021	0.02839
Maximum:	0.00515	0.00491	0.02656	0.03533
Average:	0.00464	0.00464	0.02143	0.03256
Std. dev:	0.00018	0.00011	0.00170	0.00138

The above table shows much more stable results. It also shows that GeoTIFF and GeoHEIF performances are equivalent for sequential readings of tiles of an uncompressed image on the local file system. A difference larger than the standard deviation is observed in the case of random access, with GeoTIFF appearing faster than GeoHEIF. This difference was unexpected, as the two readers were intended to be of equivalent performance. The cause was not investigated in Testbed 20.

B.4.1. Cloud optimized GeoHEIF

The following table shows the same benchmark as Table B.2, but with files read from the local host through HTTP range requests on an Apache HTTP server. The GeoHEIF and GeoTIFF readers share the same Java code for identifying the ranges of bytes containing a tile, merging consecutive ranges, sending HTTP requests and caching the result. In this benchmark, GeoTIFF appears slightly faster in sequential read operations, while GeoHEIF appears slightly faster in random read operations. Repeating the benchmark a second time produced similar results (the table below is actually the average of those two runs).

Table B.3 — Execution times (milliseconds) of image read operations through HTTP range requests

	GEOTIFF (SEQUENTIAL)	GEOHEIF (SEQUENTIAL)	GEOTIFF (RANDOM)	GEOHEIF (RANDOM)
Minimum:	0.00615	0.00648	0.08053	0.07540
Maximum:	0.00741	0.00823	0.09289	0.08957
Average:	0.00660	0.00719	0.08625	0.08289
Std. dev:	0.00028	0.00037	0.00262	0.00283

The observation that GeoHEIF is apparently slower on local file system but faster for random accesses through HTTP was unexpected. But while the difference is greater than the standard deviation, it is still close and may be a statistical fluke. The whole benchmarking process,

including relaunching the Java Virtual Machine, should be repeated at least 40 times. But it would require pushing the automation further than what was done in Testbed 20. Specialized tools such as Java Microbenchmark Harness (JMH) should be used. Nevertheless, this experiment suggests that the GeoHEIF format is as efficient as Cloud Optimized GeoTIFF for reading an image through HTTP at full resolution. The reading of multi-resolution (pyramided) images was not tested, but there is no known reason yet why the results would not be similar in that context as well.

B.5. Binding to native code

Read operations using the GDAL native library (C/C++) were also benchmarked. This benchmark used a "GDAL to Java" binding implemented with the Panama technology (the java.lang.foreign package). Panama, available since Java 22, replaced Java Native Interface (JNI) (Java 1.1). This benchmark did not use the OSGeo GDAL-Java binding, which is based on SWIG (a generator of JNI bindings). An advantage of Panama is that is provides greater control on copy operations between Java heap and native memory.

The benchmarking results are not reported for two reasons. First, the GDAL 3.9.3.0 version used for the benchmark did not include GeoHEIF support (but getting that support would have been possible by building GDAL locally), while the benchmark's goal is to compare GeoHEIF against GeoTIFF rather than to compare libraries. Second, the time measurements were unexpectedly large, which suggest that this benchmark did not used GDAL correctly. The cause may be not enabling the GDAL cache, missing or incorrect calls to `GDALRasterAdviseRead(…)`, or parameters given to `GDALRasterIO(…)` forcing more work than intended (e.g., queries too large or not aligned on tiles). The issue was not investigated in Testbed 20. Nevertheless, some observations are possible by running the benchmark application in the NetBeans profiler.

- The benchmarking application spent 38% of its time in the `GDALRasterIO(…)` function.
- The binding code (outside GDAL) spent 22% of its time in creating empty rasters. More discussion below.
- The rest of the time was spent in the benchmark application itself, mostly in matrix inversions using code that was too generic for this purpose. However, this time was not counted in the above tables.

The time spent creating empty rasters (22%) is not normal and was not observed in the pure-Java implementations of the GeoTIFF and GeoHEIF readers. The cause was not investigated and is unlikely to be related to GDAL. Those empty rasters are created by calls to Java's `Raster.createWritableRaster(SampleModel, 'Point')` method before to fill the raster's content with data provided by GDAL using strategy 3 of Figure B.1. A hypothesis for the slowness may be related to the fact that this Java method tries to use the video card memory, which is an overhead in the context of the Java-C binding.

A similar benchmark of the pure-Java GeoTIFF reader showed 8% of time in the GeoTIFF reader and 90% of time in the benchmarking application. The source of the latter inefficiency was identified as the use of an unnecessarily generic algorithm for a matrix inversion in the Apache

SIS library. This will be fixed in a future version. However, as those matrix inversions happened outside the read operations that were measured, it did not impact the time measurements reported in this annex.

C

ANNEX C (INFORMATIVE) SILVEREYE TECHNOLOGY — BENCHMARKIN CONTRIBUTIONS

ANNEX C (INFORMATIVE) SILVEREYE TECHNOLOGY — BENCHMARKING CONTRIBUTIONS

C.1. Executive Summary

Silvereye Technology contributed to Testbed 20 benchmarking in two forms:

- Provision of test data, and
- Evaluation of small tiles.

The small tiles evaluation demonstrated that high efficiency codecs, when combined with an MPEG proposed low-overhead image file format, can provide significant reduction in file size over current formats such as PNG.

C.2. Test data

As part of Silvereye Technology's contribution to Testbed 20, GeoTIFF source data was converted to HEIF, including GeoHEIF metadata. The data was Red-Green-Blue true color aerial imagery of a residential area in Canberra, Australian Capital Territory. The source data has a CC-BY license (Jacobs Group (Australia) Pty Ltd and Australian Capital Territory government). Silvereye Technology acknowledges the ACT governments Open Data policy.

Silvereye Technology converted the data using custom tools and produced several files with different encoding options (e.g., tiled or not, pyramid overview or not). This allowed benchmarking to evaluate the effect of these options under different use cases, without needing to compensate for the effects of different data.

C.3. Small tile size comparison

Silvereye Technology conducted an evaluation of the applicability of HEIF to small images, such as those used for raster tiles. This was motivated, in part, by the low-overhead image file format changes proposed in ISO/IEC 23008-12 CDAM 2.

An example of this is OpenStreetMap, where 256 pixel by 256 pixel tiles are rendered for visualization of the underlying data. The usual encoding of those tiles is as PNG images, which works well for the sharp edges expected in a raster map.

This kind of tiling is potentially the result of an OGC API for Tiles request. Since this API supports content negotiation, there is a clear path for adding HEIF-encoded files as an option where it could provide benefit.

Silvereye Technology selected two tiles from OpenStreetMap — one that is very simple (essentially a single color, from a forested area to the west of Canberra, Australian Capital Territory), and one that was more complex (multiple roads, building outline, symbols and text) from a commercial area in Canberra, Australian Capital Territory.

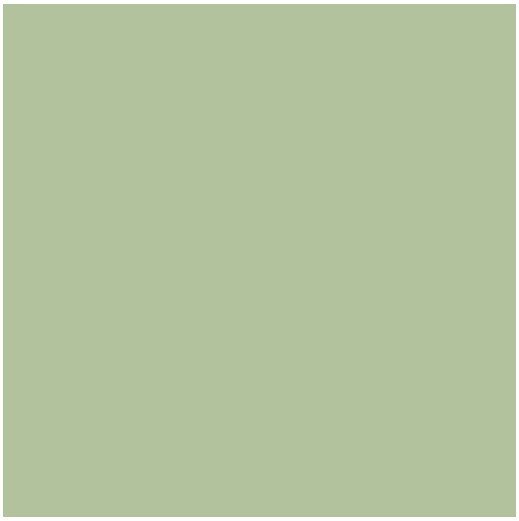

Figure C.1 — Simple OSM test tile

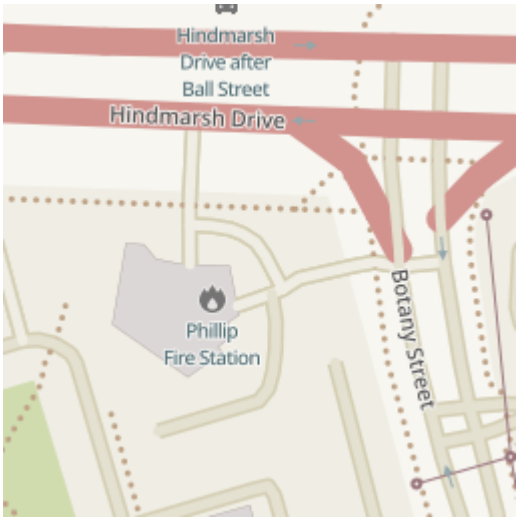

Figure C.2 — Complex OSM test tile

The tiles were compressed using the "avifenc" command line tool from libavif, with different compression quality settings (essentially trading off compression artifacts for file size). The same compression was then repeated using the experimental low-overhead option (`--mini`). In both cases the same AV1 codec was used, being the reference AOM codec.

Results are shown in the graphs below.

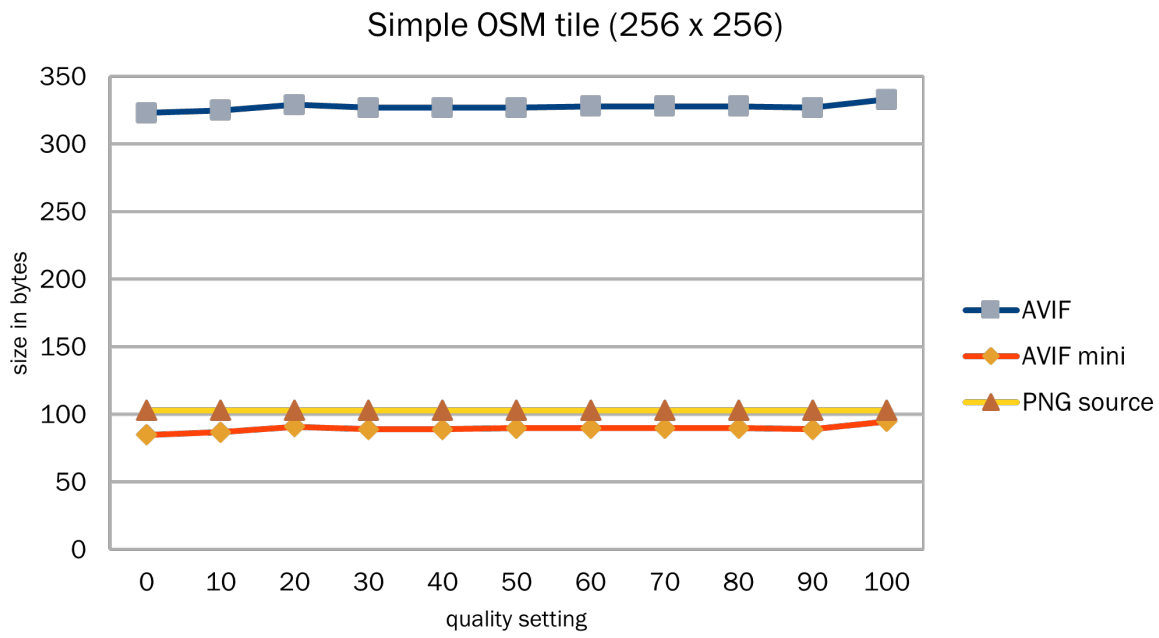

Figure C.3 — Simple tile compression size for AVIF and PNG

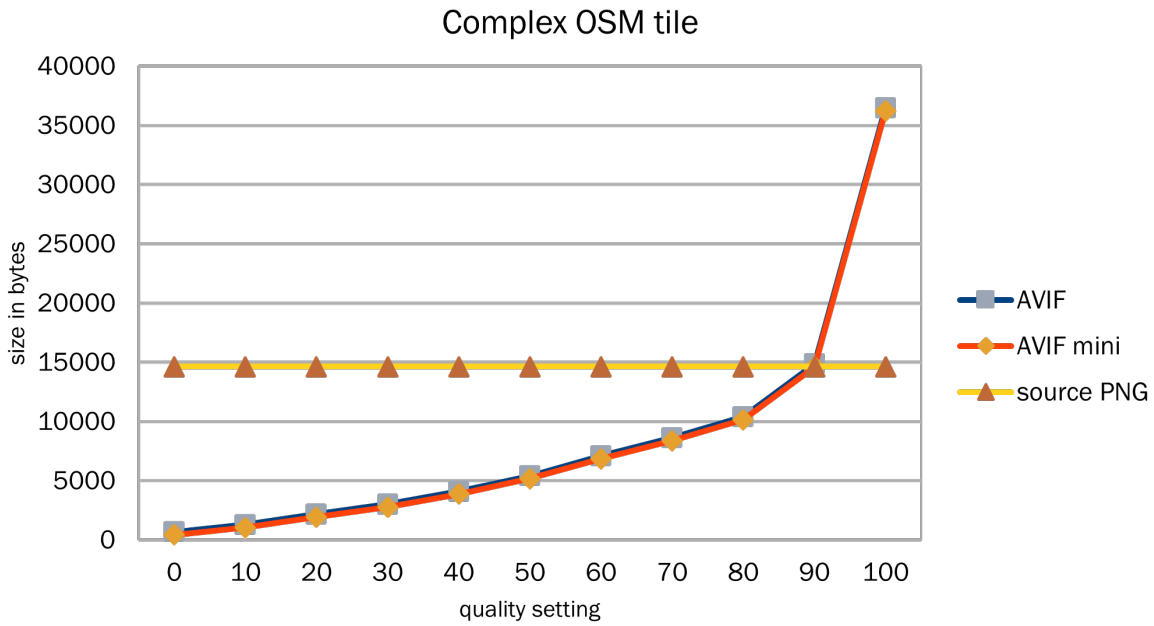

Figure C.4 — Complex tile compression size for AVIF and PNG

As a lossless compression system, there is no quality variation for PNG — it is provided in the above graphs only as a reference level.

In both cases, the low-overhead option reduces the file size by approximately 238 bytes from the full ISO/IEC 23008-12:2022 header. The relative importance of that reduction depends on the overall size of the file, being more obvious for the simple tile.

Given the simple tile has essentially only one color, it compresses very well with both PNG (which can use a single entry in the color palette), and with AV1. However the use of the low-overhead header is required to bring AVIF into a comparable file size with PNG on this simple tile content. As would be expected from the very small difference in compressed size, there is essentially no visual difference across the quality level settings.

The complex tile shows much more file size variation across the (arbitrary) quality scale, from substantially smaller than PNG to substantially larger than PNG. This is associated with much more noticeable visual differences, as shown in the samples below. Very low quality settings (0 or 10, not shown) produce distortion to the extent that the file is essentially unusable. Settings in the range 20 to 50 produce files that have some artifacts but are likely usable for many purposes. Settings above this level are visually identical to the PNG source image, at potentially half the file size.

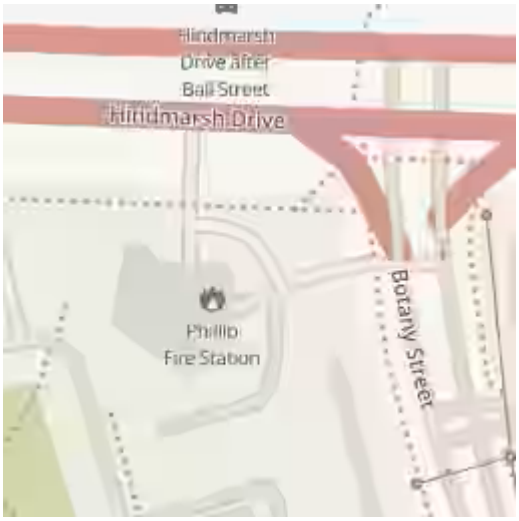

Figure C.5 — Complex tile - quality=20

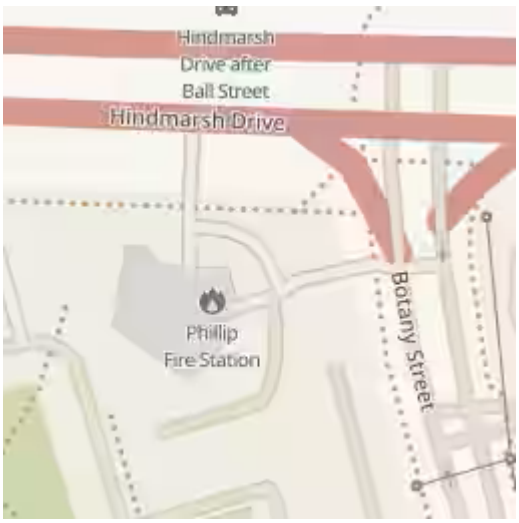

Figure C.6 — Complex tile - quality=30

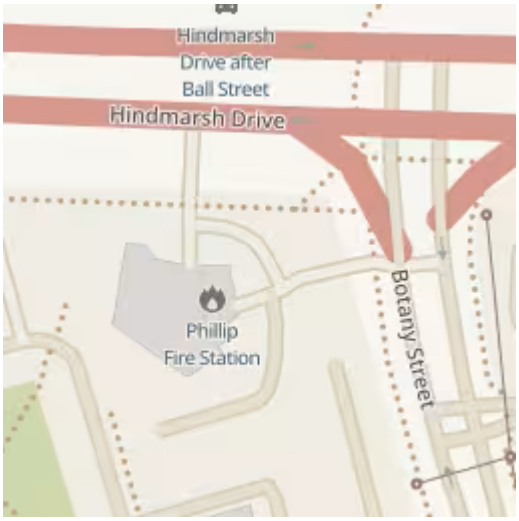

Figure C.7 — Complex tile - quality=50

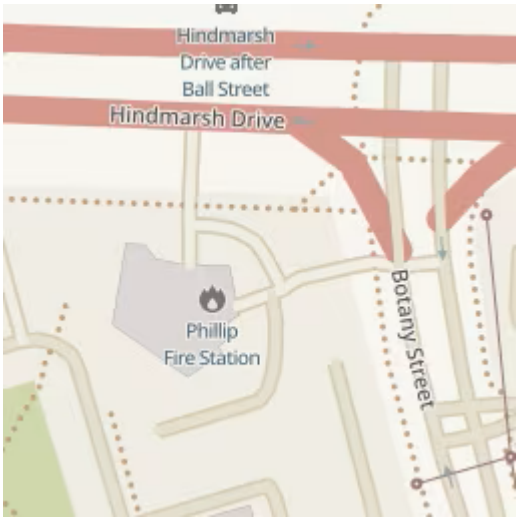

Figure C.8 — Complex tile - quality=60

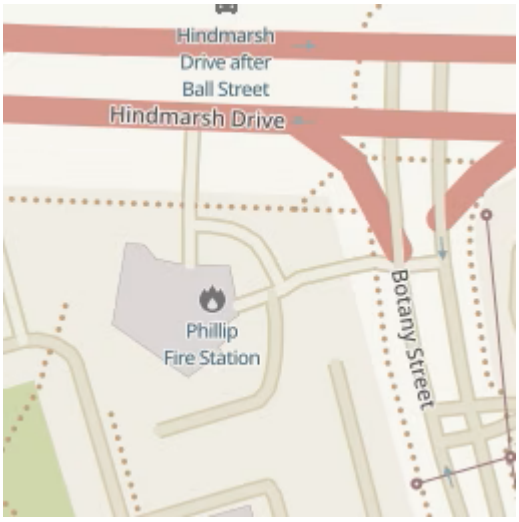

Figure C.9 — Complex tile - quality=80

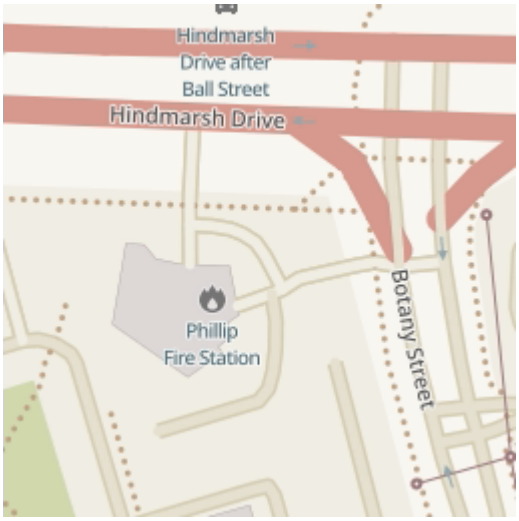

Figure C.10 — Complex tile - quality=100

By way of comparison, using an uncompressed RGB TIFF file for either test tile produces a file just under 200 kilobytes (256 x 256 x 3 = 196608 plus file overhead). Using a palette format reduces that to approximately 8 kilobytes for the simple tile, and 67 kilobytes for the complex tile.

Applying a lossless compression mechanism with TIFF helps significantly, although both PNG and AVIF are generally smaller in terms of file size:

- Simple tile, LZW compression: 329 bytes
- Simple tile, Deflate compression: 213 bytes
- Simple tile, ZSTD compression: 199 bytes
- Complex tile: LZW compression: 17706 bytes

- Complex tile: Deflate compression: 15673 bytes
- Complex tile: ZSTD compression: 15587 bytes

While a full evaluation of HEIC (i.e., H.265) was not included, an informal check suggested that it would be less compact than AV1, although still better than PNG for the complex tile example and somewhat worse than PNG for the simple tile example.

In conclusion, the use of HEIF with a high efficiency codec can produce significant file size reduction over PNG tiles, and in combination with the proposed low-overhead image file format, it is likely that it will probably be of the same order as PNG even for very simple files.

Future work could:

- Benchmark alternative compression algorithms (e.g., JPEG-2000, HEVC or EVC),
- Evaluate the effect of using larger tile sizes (e.g., 512 pixels or 1024 pixels per side),
- Provide objective evaluation of the result (e.g., PSNR, structural similarity),
- Consider other aspects such as compression effort (latency, energy consumption), and
- Evaluate applicability to other raster content (e.g., high bit depth single channel, or true color RGB).

D

ANNEX D (INFORMATIVE) HELYX – COG BENCHMARKING

ANNEX D (INFORMATIVE) HELYX — COG BENCHMARKING

Helyx were tasked with benchmarking the performance of Cloud-Optimized GeoTIFFs (COGs). Benchmarking was carried out against a range of COG datasets and creation options, and where possible, against regular TIFF and HEIF files. Because the GDAL-powered COG drivers are designed for different applications than the existing regular TIFF and HEIF drivers, there are a limited number of comparable scenarios. For example, libheif is designed to support qualitative applications (human visualization), so they have specific bit depths and band counts, and support lossy compression. COG and GeoTIFF are designed for quantitative applications (scientific analysis), so they have a wide range of bit depths, capacity for multiple bands, and support lossless compression.

D.1. Benchmarking Data

Two types of COG data were used in the benchmarking procedure: **real data** and **synthetic data**. Real data are useful for **Real Data Benchmarking** as they provide an honest representation of data values, which affects the data compression formatting variables. Common occurrences such as NoData areas outside of swaths are captured in real data. Synthetic data are useful for **Synthetic Data Benchmarking** as they remove uncertainty that arises from unpredictable data content and its effect on compression.

NOTE: This Report provides specific definitions for synthetic and real data that apply solely within this context and do not reflect universally accepted definitions of these terms. Real data derives from actual events, sensors, or user interactions, capturing the complexity of real-world systems and aiding in assessing system performance under realistic conditions. In contrast, synthetic data is artificially generated to mimic real data structures and behaviors, offering control and scalability for testing edge cases and specific scenarios.

D.1.1. Real Data

The COGs used in this benchmark represent real datasets, such as satellite images or elevation models. In Helyx's benchmarking tests an emphasis was placed on geospatial raster data over photographic raster data as this is the target of the COG format. Geospatial raster data requires a wider range of bit depths and bands as it has a broader range of data to represent (e.g., scientific imagers generating multi- and hyperspectral images, and derivatives like elevation models). Geospatial rasters may also be much larger than their photographic counterparts (in

column/row space), as they may be mosaics of several images, or generated by large sensors (such as a pushbroom scanner on an Earth Observation satellite). Photographic rasters place an emphasis on storage and transmission efficiency given their ubiquity. Unlike geospatial rasters, photographic rasters are normally used for qualitative purposes, so are less limited in compression formats (lossy compression in acceptable).

Geospatial raster data are widely available from web accessible data repositories such as the Microsoft Planetary Computer, or can be generated using GDAL binaries (gdal_translate), or other conversion softwares. Examples of real data identified for benchmarking include:

1. **Copernicus Sentinel-1 Ground Range Detected (GRD) Synthetic Aperture Radar (SAR) satellite imagery.** These data contain two bands/channels representing the dual radar polarities used during acquisition (Vertical-Transmit, Vertical-Receive, or VV and Vertical-Transmit, Horizontal-Receive, VH). Sentinel-1 data provided by Copernicus are not provided in COG format by standard so have to be converted first.

2. **NASA EO-1 Hyperion Hyperspectal satellite imagery.** These data contain 242 bands/channels, so represent an edge-case for formats like COG.

3. **USGS Landsat Near-Real-Time (NRT) multispectral imagery.** These data are dynamically retrieved by querying a Spatiotemporal Asset Catalog (STAC) and randomly downloading a recent Landsat Level-2 image.

Real data can be re-saved to COGs using GDAL, which provides parameters to change the structure of the COG. These include tiling schemes, zoom levels, overview and compression options. Testbed 20 participants were consulted as to the range of options to use to create different variants of COG datasets for benchmarking.

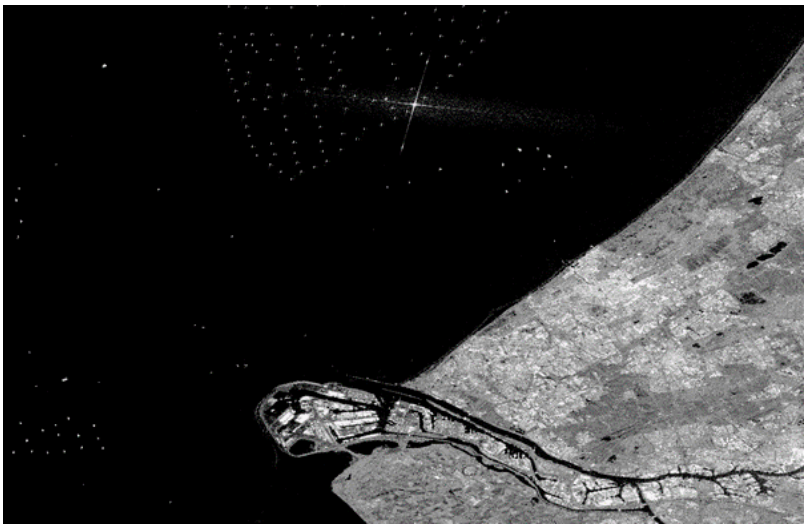

Figure D.1 — Sentinel-1 image of Rotterdam and the Southern North Sea.

D.1.2. Synthetic Data

COGs, TIFFs and HEIFs can contain procedurally generated data, either random (e.g., Gaussian noise) or formulaic (e.g., a continuous surface). These data can be generated using scientific programming tools like NumPy and written to disk using image libraries like GDAL/RasterIO (COGs) and Python Imaging Library (PIL) for TIFFs and HEIFs. Two patterns were used to generate imagery data content.

1. **Gaussian Noise.** Defined by specifying a mean and standard deviation.
2. **Mandelbrot Set.** Defined by specifying two numeric ranges and a maximum iteration value.

D.2. Benchmarking Objectives

1. **Synthetic data benchmarks** — — These tests considered the absolute performance of the COG and HEIF drivers. The tests aimed to minimize external factors that may influence test metrics, such as network latency, variations in data content (which affects compression), or complexities in driver implementation (e.g., libtiff used by GDAL, called by RasterIO in Python versus libtiff used by PIL in Python).
2. **Real data benchmarks** — These tests considered the performance of the COG and HEIF drivers in practical/"real life" scenarios. The purpose of the real data benchmarks was to ensure the synthtic data benchmark tests by either confirming them, or identifying the presence of unknown factors that affect performance (which would require further investigation).

D.3. Benchmarking Scenarios

The Testbed 20 GIMI benchmarking scenarios define the specific tests that were used to benchmark COGs and are outlined in Java code of the Scenario enumeration. The Helyx benchmarking activity aimed to follow these scenarios as closely as possible.

Each scenario modifies either a **format variable**, the **image content**, or an **operational variable**. Format variables relate to how files and their metadata are formatted. Image content relates to the data type and pattern of the raster data, as these effect compression. Operational variables relate to how the files are operated (interacted with). Scenarios are derived from a combination of possible operations and formatting options. Possible operations derived from basic read/

write and cover COG-specific operations, such as subset reading. Possible formatting options are derived from the GDAL COG driver options.

D.4. Benchmarking Approach

Helyx used Jupyter Notebooks for benchmark implementation.

- Synthetic COG and HEIF Benchmarking.ipynb* uses synthetic COGs and a local network (using Docker) to perform synthetic data benchmarking.
- Real COG Benchmarking.ipynb* uses publically accessible COGs stored on the Microsoft Planetary Computer to perform real data benchmarking. The notebooks capture metrics and generate graphs to help interpret the results.

Real benchmarking tests measured **read/write times** against pre-defined data files where there is no control over file content and creation options.

Synthetic benchmarking tests measured **read/write times** and **file sizes** against file content and creation options. Possible synthetic data creation options are as follow.

- **Pattern:**
    1. **random Gaussian noise**: This data generation process will make an image that is difficult to compress.
    2. **mandelbrot**: Fractal patterns that are quite good at modelling natural structures (e.g., topography).

- **Formats:** Either a **Rasterio/GDAL COG** or **PIL TIFF or HEIF**. COG uses Rasterio/GDAL for R/W, HEIF and basic (non-Geo) TIFF uses PIL. PIL TIFs and HEIFs have limited functionality compared to GDAL COGs (e.g., compression, overviews & block sizes don't apply).
- **Dims** (array dimensions):
    1. Specify the shape of the array (image data) to be built. Dims are specified as (Channels/Bands, Columns, Rows).
    2. HEIF currently only supports single or 3-band arrays.

- **Dtypes** (pixel data types/bit depth)
    1. These are the string flags that define bit depth and relate to the format (COG or HEIF).

2. PIL/HEIF and Rasterio use different bit depth flags, sometimes for the same thing (**Byte** in Rasterio is the same as **UInt8** in HEIF). To test, both flags must be specified.

3. Invalid combinations will simply be skipped (e.g., a 12-bit GeoTIFF, or 8-band HEIF).

- **Compression types** (GDAL/COG only)

    1. The compression algorithm to use.

- **Overviews** (GDAL/COG only)

    1. The TIFF overview levels (default is [2, 4, 8, 16]).

- **Block sizes** (GDAL/COG only)

    1. The TIFF internal block size (the size of the array "blocks" that are written to disk)

D.5. Analysis and Observations

Read and write operations were performed in a single thread.

D.5.1. Synthetic Data: Formats and Drivers

To minimise variables in assessing format and driver read/write times and file size, the following minimal creation options were specified:

- Dimensions: 3 x 512 x 512
- Bit depth: Unsigned 8-bit (Byte)
- Pattern: Random/Gaussian

A significant factor in read and write times was the compression algorithm, which differ significantly from HEIF to non-HEIF formats. To make the comparison as fair as possible the GDAL and PIL-based TIFF files were generated using JPEG compression. It is acknowledged that these are two very different technologies and the results should be interpreted as such.

D.5.1.1. Result Graphics

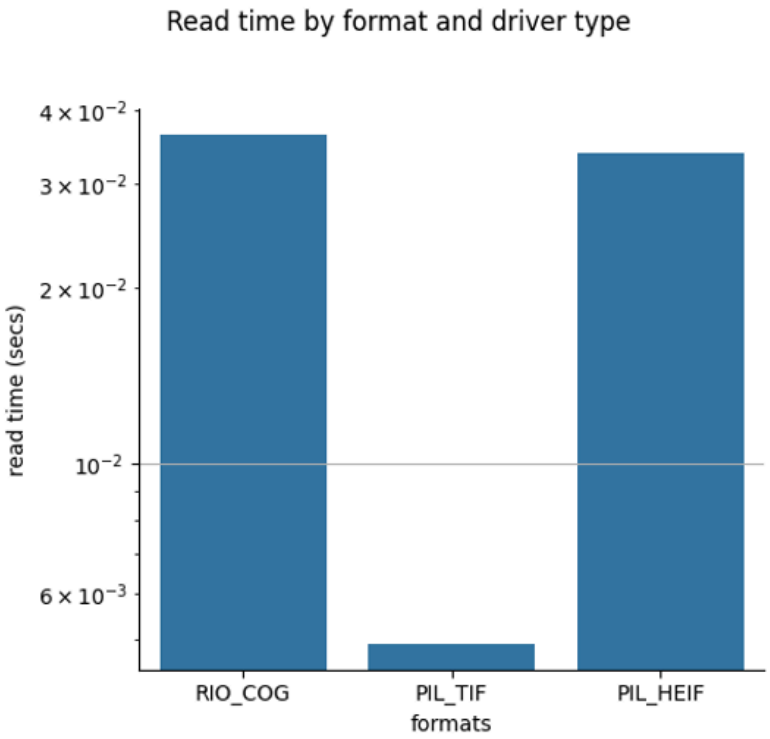

Figure D.2 — Image write times by format and drivers.

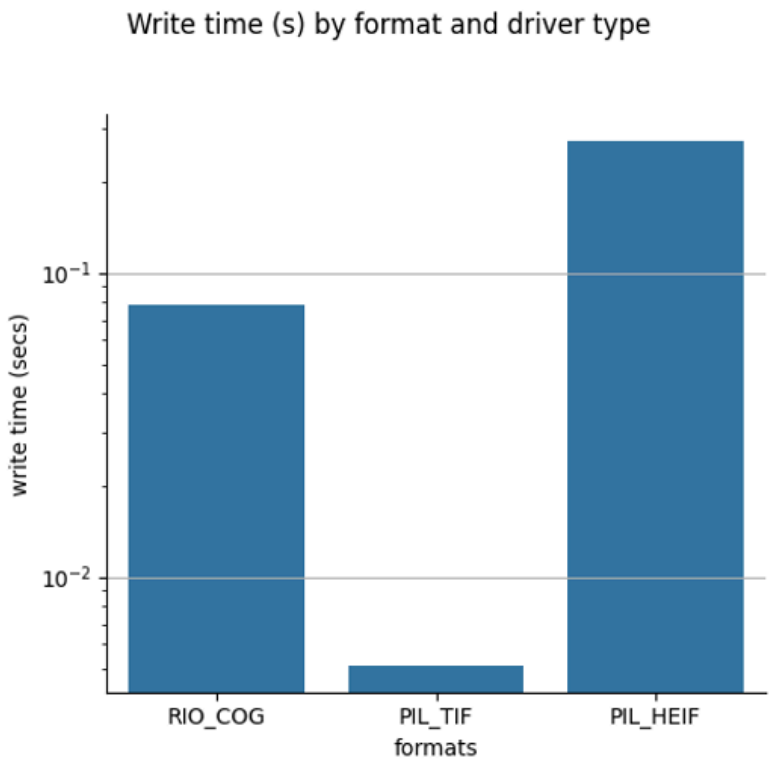

Figure D.3 — Image read times by format and drivers.

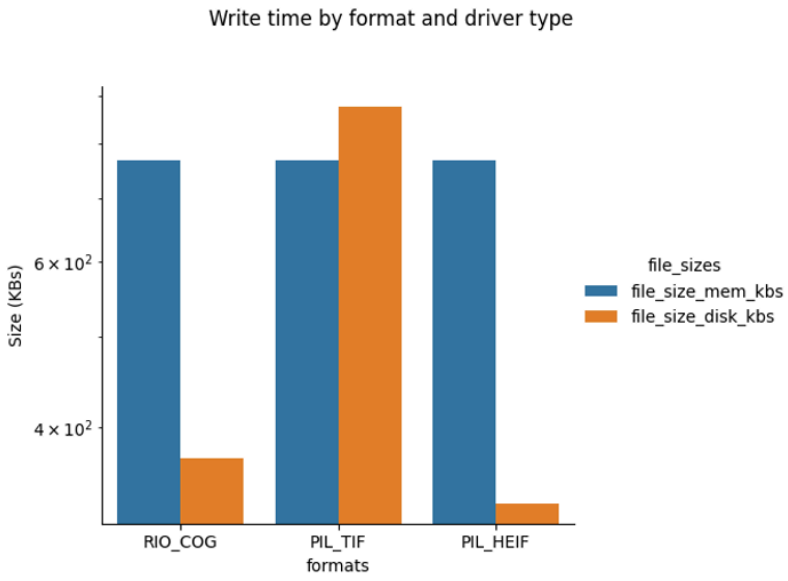

Figure D.4 — File size in memory and on disk by image format and drivers.

D.5.1.2. Findings

- PIL's TIFF format was the fasted for both read and write times.

- PIL's TIFF format was also the least efficient file size, suggesting compression is not being performed correctly, or at all. Creation options should meet the requirement for JPEG compression in a PIL TIF, see the PIL doc, but this may be user error on the author's part.
- RasterIO's COG write time is marginally slower than PILs (0.1s Vs 0.02s).
- RasterIO's COG read time is significantly slower than PILs (0.046s Vs 0.005s).
- PIL's HEIF format take considerably longer to write than the TIFF format. This is probably because of the additional computational cost of HEVC compression.
- PIL's HEIF format is the most efficient from a file size perspective.

D.5.1.3. Recommendations

- It is difficult to fairly compare the speed of HEIF to TIFF or COG because the compression algorithms are so different. A fairer assessment should also consider storage efficiency. It is likely that the computational cost (compute resource and time) of generating HEIFs balance against file size.
- In cases where speed matters most no compression, or computationally simple compression algorithms may be more suitable.

D.5.2. Synthetic Data: Image Dimensions

To minimise variables in assessing image dimensions on read/write times, the following minimal creation options were specified:

- Formats: PIL_HEIF | PIL_TIF | RIO_COG
- Dimensions: 1|3 x 512|1024|4096 x 512|1024|2048
- Blocksize: 256 x 256
- Bit depth: Unsigned 8-bit (Byte)
- Pattern: Random/Gaussian

D.5.2.1. Result Graphics

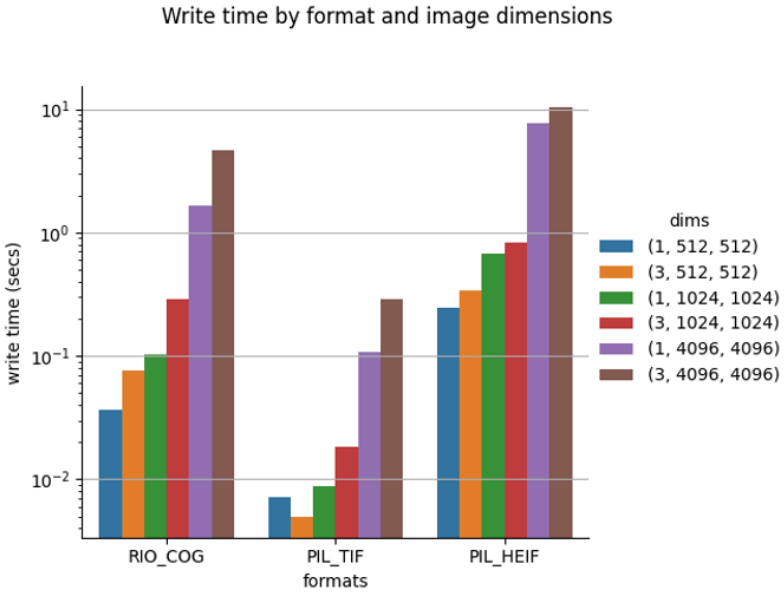

Figure D.5 — Image write times by format and dimensions.

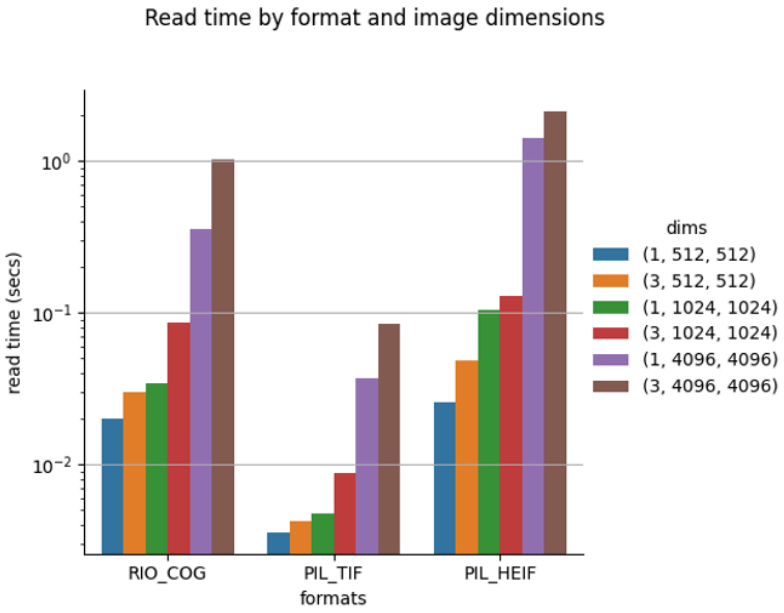

Figure D.6 — Image read times by format and dimensions.

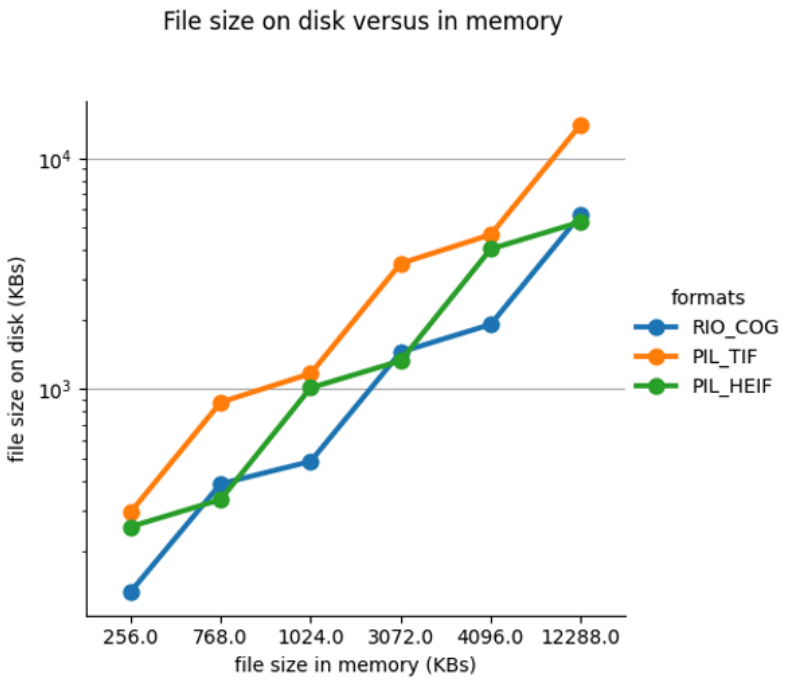

Figure D.7 — Image file size in memory (uncompressed) and on disk (compressed) across a range of image sizes.

D.5.2.2. Findings

- Read/write times have an exponential correlation to image dimensions. Smaller images are written and read exponentially faster than larger images.
- File sizes in memory (uncompressed) compared to on disk (compressed using JPEG for TIF/COG, and HEVC for HEIF) have an unexpected relationship. Generally HEIF is the most efficient, however in some cases RasterIO's TIFF with JPEG compression is better (3072 KBs in memory, 3×1024x1024 shape). This may be because quality is set to 100% and the image is random, meaning there is little any compression algorithm can improve.

D.5.2.3. Recommendations

- Investigating multi-threaded read/write times for larger images would be useful to reflect real-world scenarios. This will probably have an impact on the time it takes to perform more complex compression/decompression.
- Investigating the effect of lossy compression would be beneficial as this is critical to qualitative image processing workflows (rendering basemaps etc). It is less relevant for scientific images where lossless compression is necessary.

D.5.3. Synthetic Data: Bit Depths

To minimise variables in assessing bit depths on read/write times, the following minimal creation options were specified:

- Formats: PIL_HEIF | PIL_TIF | RIO_COG
- Dimensions: 3 x 1024 x 1024
- Blocksize: 256 x 256
- Bit depth: Unsigned 8-bit (Byte) | Unsigned 10-bit | Unsigned 12-bit | Unsigned 16-bit | Signed 16-bit | Unsigned 32-bit | Signed 32-bit | Float 32-bit | Float 64-bit | Complex 64-bit
- Pattern: Random/Gaussian

D.5.3.1. Result Graphics

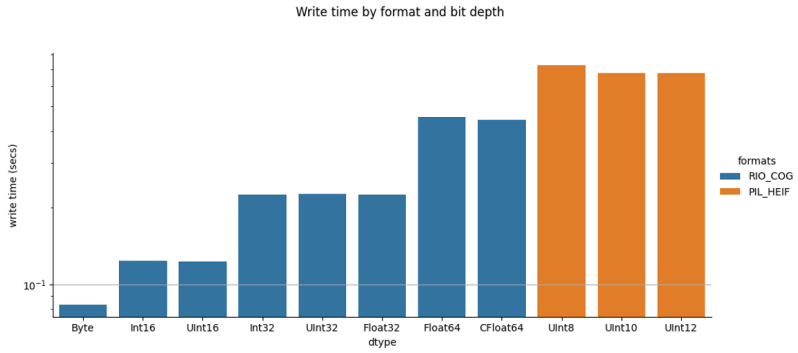

Figure D.8 — Image write times by format and bit depth.

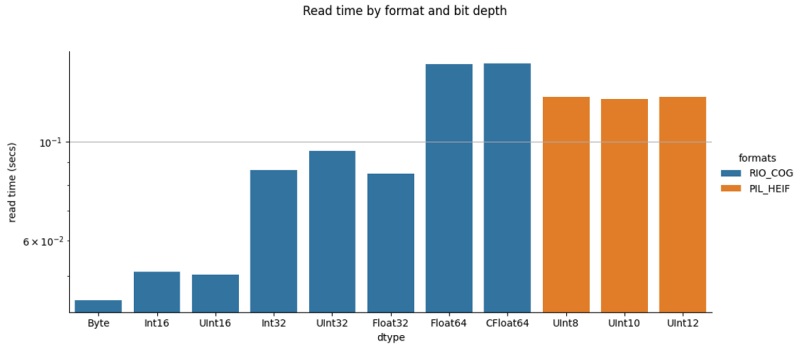

Figure D.9 — Image read times by format and bit depth.

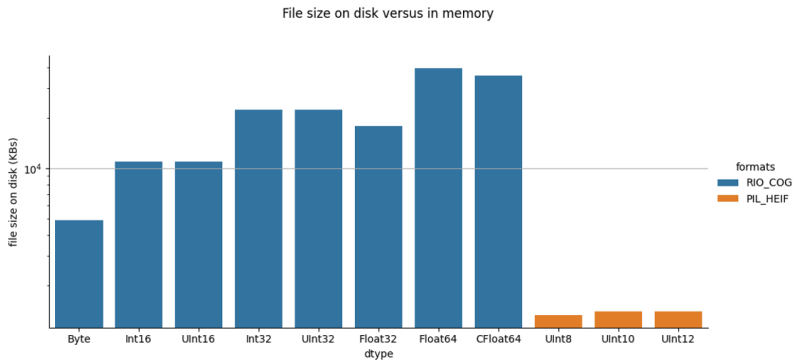

Figure D.10 — Image file size in memory (uncompressed) and on disk (compressed) across a range of bit depths.

D.5.3.2. Findings

- Read and write times are proportional to bit depth for the GDAL-generated COGs. 8-Bit images are read and written the fastest with Complex and floating point images being the slowest.
- Read and write times are similar for the HEIF images, so not proportional to bit depth.
- 64-bit floating point COG images took slightly longer to write than 64-bit complex floating point COG images, which wasn't expected.
- HEIF was only comparable at 8-bit/Byte for a 3-band image. In this case it was much slower to read and write than COGs, but produced much smaller files.
- Tests were constrained by Numpy's support for bit depths. Numpy covers most integer and float bit depths used by HEIF and COG. However for complex numbers only supports 64-bit.

D.5.3.3. Recommendations

- HEIF's main advantages in file size are reduced when addressing quantitative image processing operations (high bit depths, lossless compression).

D.5.4. Synthetic Data: Bit Depth and Compression

To minimise variables in assessing bit depth and compression on read/write times, the following minimal creation options were specified:

- Formats: PIL_HEIF | PIL_TIF | RIO_COG
- Dimensions: 3 x 1024 x 1024

- Blocksize: 256 x 256
- Bit depth: Unsigned 8-bit (Byte) | Unsigned 10-bit | Unsigned 12-bit | Unsigned 16-bit | Signed 16-bit | Unsigned 32-bit | Signed 32-bit | Float 32-bit | Float 64-bit | Complex 64-bit
- Compression: None | LZW | Deflate | LERC | JPEG | HEVC
- Pattern: Random/Gaussian

D.5.4.1. Result Graphics

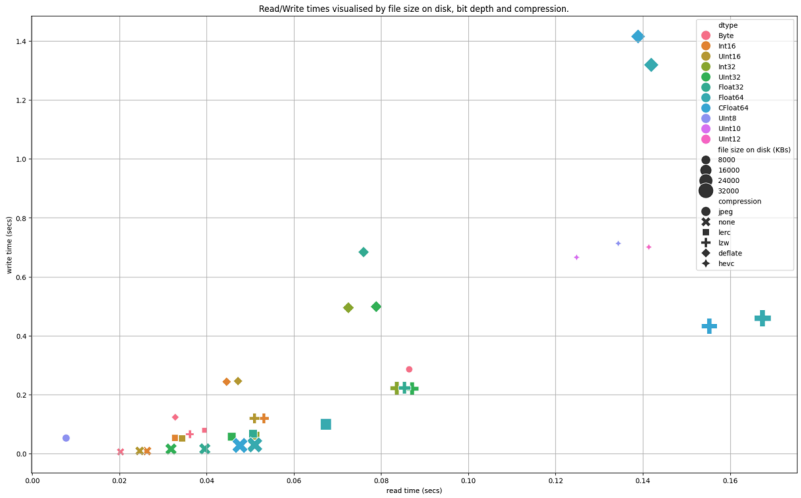

Figure D.11 — Image read and write times, and size by bit depth and compression. Marker size represents size on disk, colour represents bit depth and marker shape represents compression type.

D.5.4.2. Findings

- HEIF read/write times and file sizes are consistent, irrespective of bit depth.
- COG read/write times and file sizes vary by bit depth and compression.
- COG file sizes do not correlate to bit depth but do correlate with compression.
- No compression is the fastest to write, but 8-bit JPEGs are faster to read.

D.5.4.3. Recommendations

- Benchmarking did not consider lossless compression, where the HEVC algorithm is likely to excel. Running additional benchmarks on lossy compression is recommended.

D.5.5. Real Data

The real data benchmarking workflow is as follows.

1. Query a STAC server using a dynamic query (e.g., a period of time from present). This will improve the likelihood different images are downloaded each time the benchmark is run. High cloud images are filtered out as these will have reduced image content. The Landsat-C2-L2 collection from the Microsoft Planetary Computer is used in this benchmark. Images with 10% or less cloud acquired within 2 days of the current time (when the test was run) are selected.

2. Select a random set of items returned by the query. This further reduces the likelihood of the same images being downloaded. In this case, 10 images were selected.

3. For each STAC item, iterate a set of assets and download the assets. Time how long it takes to download the asset, how long it takes to save the asset locally to disk, and how large the asset is on local disk. The red, green and blue Landsat bands were used as they are all of the same resolution and have similar spectral properties (so will contain similar information).

All available image footprints are shown in grey, footprints randomly selected for download are shown in red. image::../images/stac_item_footprints.png[align=center, width=400]

D.5.5.1. Result Graphics

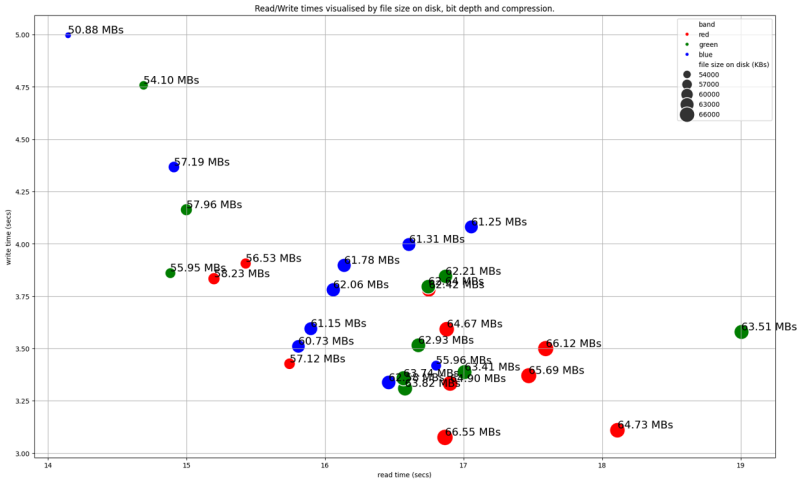

Figure D.12 — A scatterplot showing the relationship between read/retrieve time of STAC items over the open internet (into memory) and the time taken to write them locally from memory to disk. Marker colours define assets (bands), marker sizes define local file size on disk.

D.5.5.2. Findings

- As expected, there is no clear relationship between read/write time or file size when retrieving STAC items over the open internet.
- There is a very weak pattern between asset type, read time and file size. Red assets take slightly longer to read and are slightly larger. Blue assets are slightly quicker to read and are slightly smaller. Green assets are in the middle speed/size wise. There are many factors that could affect this, from how images are selected from the STAC server (low cloud images tend to be over arid environments), to the way Landsat is configured, to how sunlight is reflected by the ground.

D.5.5.3. Recommendations

- More thorough benchmarking will probably show that read times are subject to broad external factors, such as overall internet activity.
- Comparing a geo- and cloud-enabled HEIF format would be more meaningful than an isolated assessment of COGs, although should still be performed in a rigorous manner (e.g., over a longer time period) to isolate time-variant factors (like internet traffic).

D.5.6. General Observations

- **Imagery products versus imagery data.** Imagery products, like those generated by satellites, require rich metadata for correct use. Examples of this are the CEOS format, ESA's SAFE format, and Airbus's DIMAP format. Practically, satellite image products contain multiple files in a range of formats. This means products cannot simply be converted into COG as the format lacks the flexibility to capture all the required metadata. Products are also generated at "data levels" to support different use cases. Lower-level products are less processed, meaning they can support a greater range of uses, but require more processing and expertise. Higher level products are easier to use, but more limited in application. Imagery products are more suited to being represented as a file (like a COG) at higher processing levels where complex metadata have been accounted for. Examples of complex metadata are sensor models (Rational Polynomial Coefficients, Ground Control Points) and calibration co-efficients. These often need to be interpreted in context of the full image (not a subset).
- **Limited geospatial and scientific support for HEIF in Python.** It was only possible to benchmark HEIF using the Python Imaging Library (PIL) HEIF wrapper functions, which lacks the rich geospatial and scientific functionality of GDAL (accessed via RasterIO). The fundamental differences between current HEIF drivers and COG drivers are significant, based on their intended application (qualitative, media versus quantitative analysis, respectively). It would be worthwhile to re-benchmark HEIF once a Geo-enabled version of it is available.

- **Difference in formatting options (COG and HEIF).** COG and HEIF do not support the same formatting options, specifically bit depth, so can only be compared where they do.

D.5.7. Future Work

- Benchmark a geo-enabled HEIF driver against the current COG drivers once one is available. Performing both synthetic data (on a closed network) and real data (on an open network) would be interesting.
- Further benchmarking of lossy compression algorithms would be worthwhile to more thoroughly understand the benefits of HEIF/HEVC.
- A more rigorous investigation into real data on an open network is needed as there wasn't real time to go into this in the course of Testbed 20 GIMI.