CCSDS 122.0-B-2

Kasper Skog

27 Jun 2024

Contents

1 Introduction

1.1 Scope

This document describes the setup, implementation details and testing of the CCSDS (Consultative Comittee for Space Data Systems) 122.0-B-2 image compression algorithm (AD1). Previous knowledge of the algorithm is expected. The CCSDS standard, linked in the 'Further Reading' section 1.3, contains a more in-depth look at the inner workings of the algorithm and mathematics.

This implementation was created as a tailored example of the CCSDS standard for the CI (Comet Interceptor) mission. Furthermore, the motivation to create an in-house version of the algorithm was to increase characterization accuracy. This was achieved by using data, software and hardware comparable to their counterparts used during the final mission. Non-vital parts of the algorithm were discarded, thus decreasing computational complexity. This reduced the memory footprint and power requirements to a level lower than other available implementations.

The main purpose of this implementation is to produce losslessly compressed files from 1024x1024 pixels and high aspect ratio grayscale images and to characterize the compression performance in an embedded environment.

1.2 Abbreviations and Acronyms

CCSDS	Consultative Comittee for Space Data Systems
CI	Comet Interceptor
DWT	Discrete Wavelet Transform
RAM	Random Access Memory
VENV	Virtual Environment
CSV	Comma Separated Values
AUR	Arch User Repository
ROM	Read Only Memory
ELF	Executable and Linkable Format
SWD	Serial Wire Debug
GDB	Gnu Debugger
CRC	Cyclic Redundancy Check
EDBG	Embedded Debugger
CPU	Central Processing Unit

1.3 Further Reading

1. CCSDS 122.0-B-2 Reference standard
2. CCSDS 120.1-G-3 Informational report
3. University of Nebraska CCSDS
4. White Dward 2.0
5. SAM V71 Xplained Ultra evaluation kit product page
6. SEGGER website for JLink

2 Implementation

2.1 CCSDS 122.0 basics

The algorithm is based on the wavelet transform, which like the Fourier transform, produces a set of coefficients containing frequency information. This process is performed for a total of three times, creating a tree-like structure. The tree-like structure contains significant redundancy with high correlation between coefficients. These coefficients are then encoded bitplane by bitplane while discarding any bits which can be derived from previously encoded data.

The input data used must be a binary file containing raw pixel data of one color channel. Each pixel must be represented by two bytes. Information about the size of the image and the bitdepth of the data are given as command line arguments when running the algorithm.

2.2 Key Considerations

This implementation does not include the option for a floating point DWT (Discrete Wavelet Transform) or a segment byte limit, as these options produce a lossy compressed file. However, these can be added without exessive alteration of existing program logic. Note that lossy compression is still possible using debug options that stop the compression process prematurely.

There is no decompression logic as it is not performed in an embedded environment nor is it resource limited. Instead, the generated files should be decompressed with an external decompressor. Both the university of Nebraska's reference implementation (AD3) and the white dwarf 2.0 (AD4) provided by CCSDS are recommended for cross-validation. Both are readily available on the internet, although white dwarf requires the creation of a free account. Links can be found in chapter 1.3.

Unlike the reference standard states, the input data must be 16-bit raw pixel data with bitdepths between 0 and 12 (inclusive). Note that this means that each pixel value must occupy two bytes even with 8-bit images. This is important to keep in mind when assessing the compression ratio between files. Input data was determined to not need a larger range and the internal data representation was changed to reflect this.

The image dimensions must have a width between 17 and 2^{20} pixels. The height of the image must be at least 17, but there is no upper limit.

A further improvement to consider in the future is the ability to stream in data instead of reading from a file. This would reduce the amount of RAM

(Random Access Memory) needed to compress large images. The CCSDS informational report (AD2), includes a proposed implementation of this logic. A full reworking of the DWT would then be required.

2.3 Project Structure

The project is partitioned to five directories: core, devkit, python, res and tests. Since the resource directory is large, it has been zipped and stored separately.

2.3.1 Core

This directory contains all of the C sources for the algorithm and all file reading logic. The `wavelet_transform.c` file includes a functional reverse wavelet transform, but it should not be included in any mission binaries. It is only used by the legacy python implementation for increased performance.

2.3.2 Devkit

Contains necessary driver files and communication logic to compress images on the `SAMV71 Xplained Ultra` development kit. File IO (input/output) has different implementations depending on the environment. These are compiled conditionally based on the `EMBEDDED` flag.

2.3.3 Python

Contains the first, now out-of-date, implementation of the compression algorithm. It requires a cython library and uses the C sources for the DWT. All files in this folder can be considered legacy code and discarded. Some tests use the `rccsds.py` decoder, but it does not support multiple segments. All comparisons should be migrated to use the aforementioned third party decoders.

2.3.4 Res

As the name suggests, this directory contains all image resources used for tests. The set of test images have been collected from the official NASA website, OPIC and EnVisS teams. Note that the NASA images are enhanced for the public and might not represent actual image data from a mission. These images contain asteroids at different distances and starry backgrounds with and without comets. Furthermore, no reliable data exists about exposure times or the sensors used for these images. The OPIC and EnVisS images are still preliminary and more accurate images are required. This means that the algorithm can not be accurately characterized for true mission performance before receiving better data.

2.3.5 Tests

Includes python scripts for testing compression ratio, performance when compressing strip images and plotting generated data. In addition, some tools for generating raw data from different image formats are included.

2.4 Installation

All following instructions are inteded for linux environments. In order to setup things according to current (2024) Python guidelines, a venv (virtual environment) is required.

2.4.1 Setup virtual environment

1. `python -m venv YOUR_ENV_NAME --system-site-packages`

   This creates a folder named YOUR_ENV_NAME and adds all venv related files inside. The '–system-site-packages' flag reuses already existing packages.

2. `cd YOUR_ENV_NAME`

3. `pip install pillow numpy numba matplotlib setuptools cython serial pyserial`

   This installs all required python packages.

2.4.2 Setup project structure and files

1. `clone project files to YOUR_ENV_NAME`
2. `cd YOUR_ENV_NAME/ccsds-122.0-B-2`

   From this point forward all commands are written as they would be run in the project directory 'ccsds-122.0-B-2'.

3. clone 'res' directory (maintained separately) to your newly created project directory.

4. `(cd tests; python bmp_to_raw.py)`

   This converts all images in 'res' to raw files used by the compressor. Changing the root folder for recursive search is possible in the python file. Remember to do this when adding a directory of images! The parenthesis make the console launch a sub-process and return to the original path after completing the command.

5. `(cd python/cython; python compile.py)`

   This generates the c_dwt library to use the DWT in python directly. Changing the C files does not trigger auto generation. Therefore, this step must be done each time any DWT code is changed.

6. `make`

   Builds the 'ccsds.bin' binary in the ./build directory.

The project is now ready for testing! Your project structure should now look like the following:

```
.
|-- build
|-- core
|   |-- include
|   `-- src
|-- devkit
|   |-- build
|   |-- core
|   |   |-- drivers
|   |   |   |-- CMSIS
|   |   |   |-- config
|   |   |   |-- hal
|   |   |   |-- hpl
|   |   |   `-- hri
|   |   |-- samv71b
|   |   |   |-- gcc
|   |   |   `-- include
|   |   `-- src
|   `-- python
|-- python
|   `-- cython
|-- res
|   |-- enviss
|   |   |-- raw
|   |   `-- txt
|   |-- noise
|   |   `-- raw
|   |-- opic
|   |   |-- oraw
|   |   `-- raw
|   |-- pattern
|   |   `-- raw
|   `-- space
|       `-- raw
`-- tests
```

3 Testing

3.1 PC

On PC testing is quite simple. The compiled binary `ccsds.bin` can be called with four command line arguments: width, height, bitdepth and the input file name. This is, in essence, all that is needed to verify the copmressor perfomance. However, some testing scripts are provided to facilitate easy reproducibility.

Most scripts can be run without external tools and none require external devices. There are tests to verify losslessness and compression ratio. These tests output results to csv (Comma Separated Values) file that can be plotted with `plot.py`. Since there is no decoder, one must provide the python file with a path to the decoder in order to run `test_random_c.py`. In addition, the test must be modified to correctly call the chosen decoder. Note that this is the only test that verfies that the resulting '.cmp' files can be decoded.

`test_compression_ratio.py` goes through every raw file it finds recursively starting from a 'root' folder set by the user. In addition to the result.csv file, the compression time and compression ratio is displayed in the console for each image.

`test_strips.py`, with similar recursive search, produces a user set number of horizontal strip images for each source image. These strips have a set height that can be adjusten in the python file. After all strips generated from an image are compressed, the results are averaged and displayed to the user.

`test_random_c.py` compresses the recursively searched images and tries to decompress them with the provided decompressor. An error is produced and the test stopped if the decompressed file does not match the original.

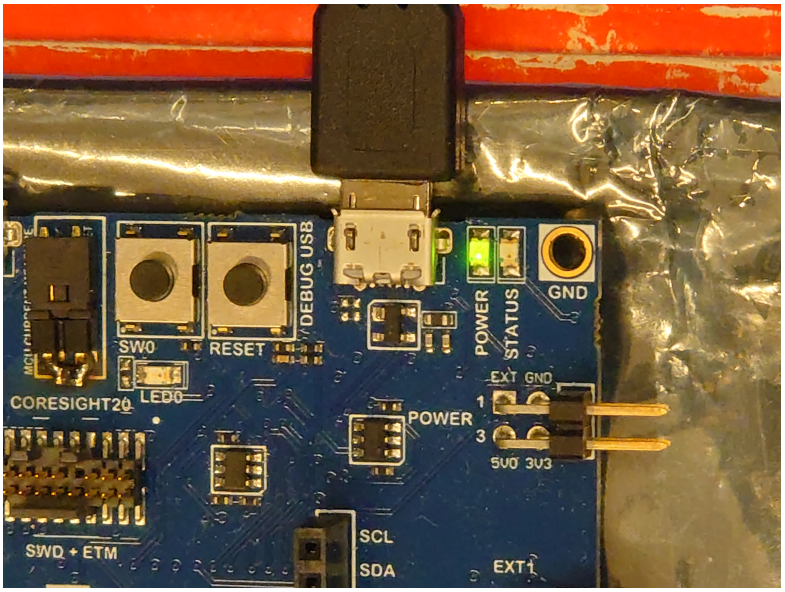
Addition of test images is performed by adding a folder of '.bmp' files to the 'res' folder and running `bmp_to_raw.py`. `OPICDecoder.py` must be used if the images are of the format '.oraw' produced by the OPIC team.

Any additional scripts or tests should be placed in this directory. `common_testing.py` contains the colors used in all tests. This is also where useful math operations are recommended to be stored.

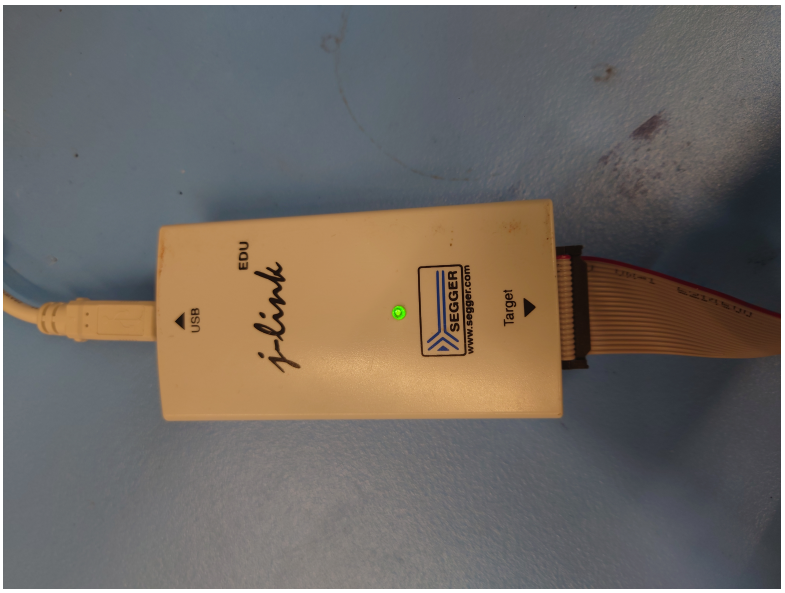
3.2 Embedded

The interface for flashing and debugging the SAMV71 Xplained Ultra devkit requires the installation of JLink tools. These can be found in the AUR (Arch User Repository) or SEGGER's website. After installation, setup should be as easy as:

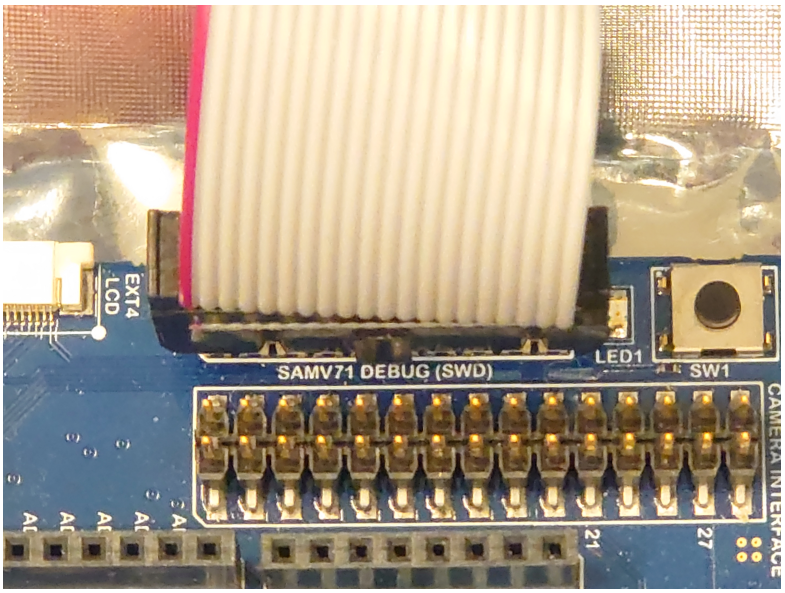
1. `connect usb to 'DEBUG USB' slot on the devkit`



2. `connect usb and ribbon cable to JLink device`

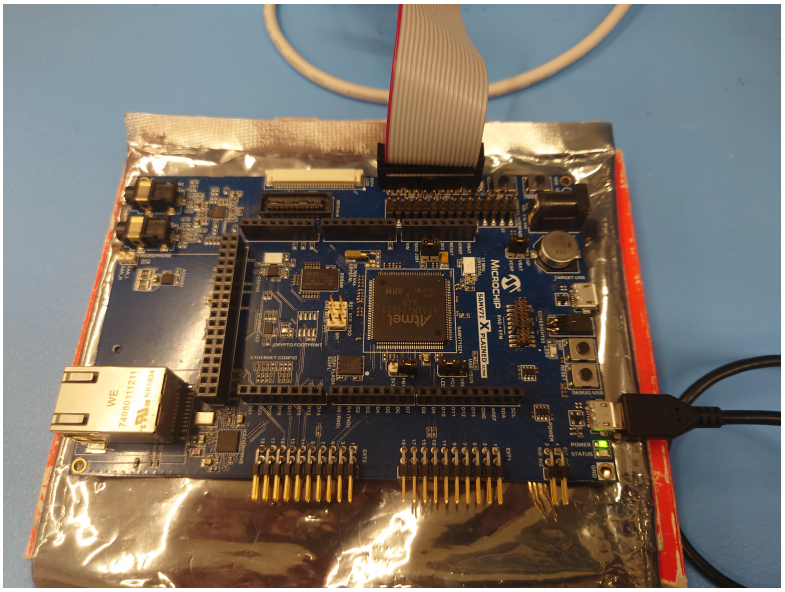


3. `connect ribbon cable to devkit`



Notice that the notch is towards the board.

After all physical connections the board should look like the following:



4. `cd devkit`

5. `make`

   Builds the binary and ELF(Executable and Linkable Format) file for the devkit and displays the amount of memory allocated from ROM (Read Only Memory) and RAM.

6. `make flash`

   Uses JLinkExe to flash the board through SWD (Serial Wire Debug). Calls make all internally.

7. (optional) `make debug`

   Launches a JLinkGDBServer and connects GDB (Gnu Debugger) to it. No direct state is read from the devkit. Instead, the compiled ELF and flashed binary files are stepped in synch. These files must match in order to debug correctly.

After setup the python interface found in `project_root/devkit/python/io_test.py` can be launched. This script is used to send raw image data and receive compressed files. Internally the files are divided into packets and sent with a corresponding CRC (Cyclic Redundancy Check). The packets are resent if the payload is corrupted. Thankfully, the devkit's EDBG (Embedded Debugger) is very reliable and resending is therefore seldom required.

The script automatically searches the host system's serial ports for the one named "EDBG" and connects to it. After a succesful connection the prompt opens. Messages incomming to the devkit are prefixed with `[I]` and outgoing with `[O]`.

Example communication of uploading a file, compressing it and receiving the result:

```
Connected: yes, port: /dev/ttyACM0, baud: 115200
[I]>> test
b'\n\x00\x01\x02\x03\x04\x05\x06\x07<'
[I]>> file
[I]>> Input file:../../res/noise/raw/test_image_noise_128.raw
[I]>> width, height, bitdepth:128 128 8
[I]>> Sending file
[O]>> received packet 512/512
[I]>> compress
[O]>> received 13 packets (767 B)
[O]>> 542.271728515625 ms
[I]>> q
```

The `test` command is optional and used to verify that input is read correctly. An file is chosen with the `file` command. This command launches a prompt where the filepath is entered. Note that image choice is limited due to the amount of available memory (384 KB). This effectively means that the largest compressable image is 256x256 pixels.

After the file has succesfully been sent, the compression is started with `compress`. This command does three things. First, it launches an internal timer. Second, it sends a packet that starts the compression via the EDBG. Finally, the compressed image is saved and the timer stopped. The script then displays the compression time in milliseconds. The compressed file is saved to `output.cmp`. This file can now be decompressed with any decompressor compliant with the reference standard.

4 Results

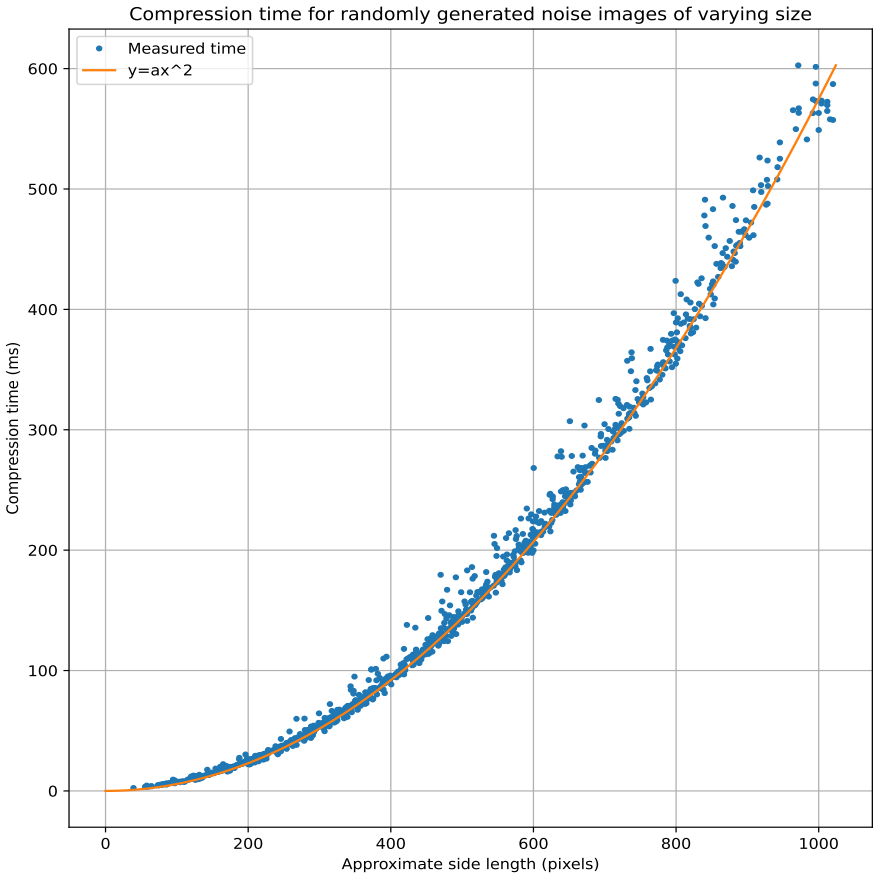


Figure 1: Compression time of randomly generated rectangular noise images with width and height ranging from 0 to 1024

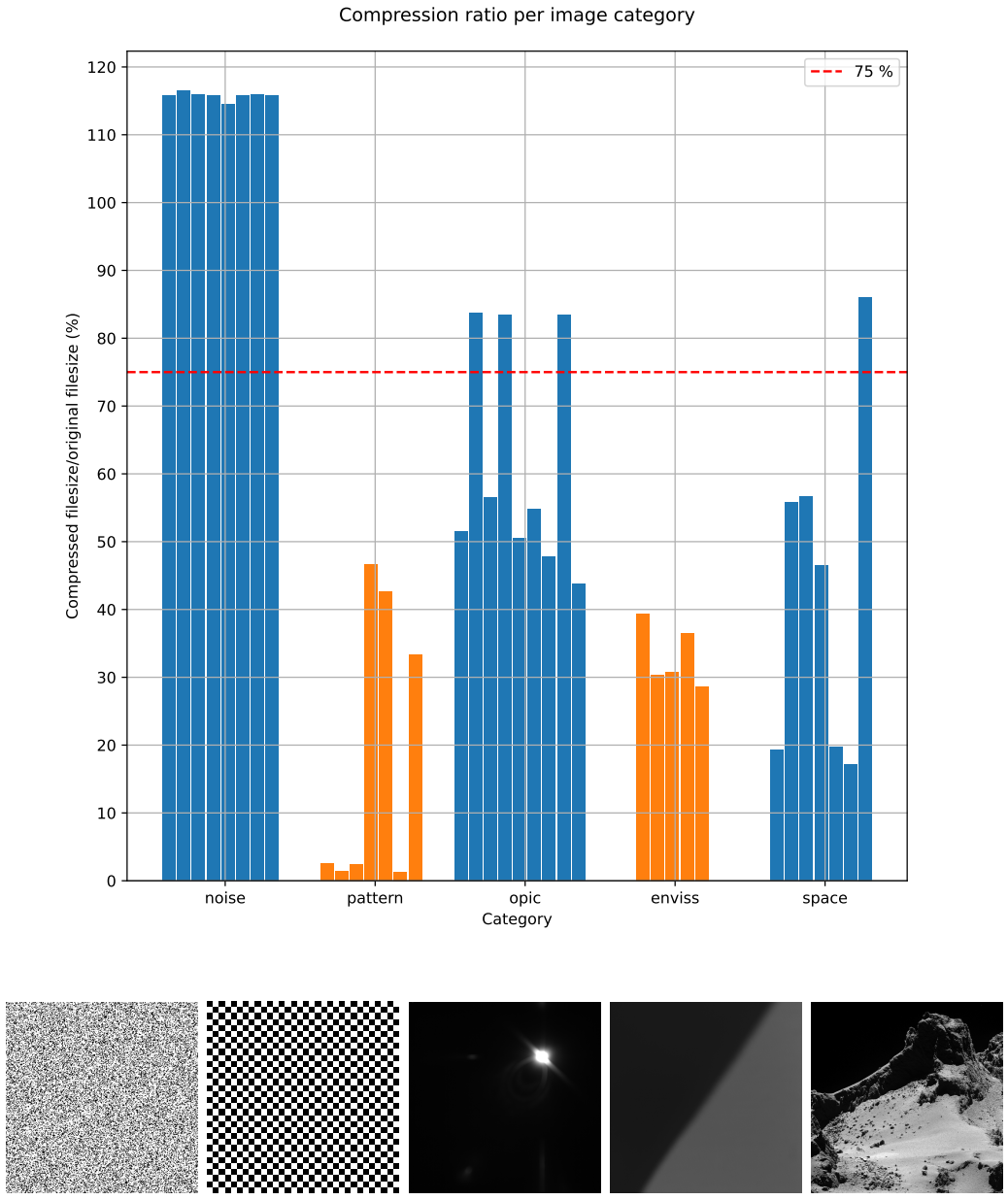


Figure 2: Compression ratio for different categories of natural and generated images with examples for: noise, pattern, opic, enviss and space

5 Conclusions

On a modern CPU (Central Processing Unit) with a clock speed of 3.9 GHz, compression of the largest images 1024x1024 takes around 600 ms. This is shown in figure 1, which presents the worst case scenario of compressing pure white noise. Note that the compression time is directly proportional to the number of pixels in the image, whereas the aspect ratio has little effect on speed. Moreover, the implementation is single-threaded due to the limited processing power of the devkit. This means that the performance of the devkit can be estimated based on clock speed alone.

Figure 2 shows that the compression ratio is highly dependent on the amount of fine grain detail in the image. Pure noise does not compress at all since the compression is based on coherence or entropy of the image. Noise from an imaging sensor affects the compression in a similar way. This can be observed in the `OPIC` and `Space` categories of figure 2. Images in the `pattern` and `EnVisS` categories consist of repeating and uniform patterns. These should and do compress well, meaning below the minimum compression limit of 75%. Whenever possible, images should be filtered before compression to reduce the effect of noise. In the case of sensor noise, the noise is time invariant and can be adjusted for with a simple offset.

The implementation works well on a wide range of images. Due to its lower computational complexity, it is light weight and thus it can run on resource bounded hardware. Moreover, it does not significantly compromise compression ratio while maintaining a small memory footprint. Future improvements should focus on procuring more accurate test data and fine-tuning compression parameters.