Journal of Engineering Science, Vol. 18(1), 1–16, 2022

Parallelization of CCSDS Hyperspectral Image Compression Using OpenMP

Nur Ashiqin Nur Shaharim[1], Tan Lit Chez[1], Ahmad Zarif Zainul[1] & Nor Rizuan Mat Noor[1*]

[1]School of Electrical & Electronic Engineering, Universiti Sains Malaysia, 14300, Penang, Malaysia

[*]Corresponding author: nrmn@usm.my

Abstract: *The Consultative Committee for Space Data Systems (CCSDS) has released the Lossless Multispectral and Hyperspectral Image Compression standard (CCSDS-MHC, also referred to as CCSDS-123) as the standard for lossless compressing the hyperspectral images taken by spaceborne/airborne imagers. Currently, most implementations of the CCSDS-MHC algorithm utilize a single processor thread for the compression process. However, CCSDS-MHC has the potential to operate on multi-threaded systems with the use of parallelization. The introduction of multi-threaded processing systems on space satellites could further decrease the execution time of the system. In this research, the aim is to design a parallelization algorithm for CCSDS-MHC using OpenMP. The first step of the research is converting the CCSDS-MHC algorithm into a full programme in C/C++ with both compression and decompression features. Next, the parallelizable section of the algorithm is identified and coded using OpenMP. The algorithm is parallelized by dividing the bands of hyperspectral images into several continuous chunks and running them concurrently. The program is then tested in several systems with different numbers of threads. The execution of the parallelized CCSDS-MHC algorithm shows significant speedups for all the systems and hyperspectral images tested.*

Keywords: Lossless hyperspectral image compression, CCSDS-123, OpenMP, Multicore processor

1. INTRODUCTION

Remote sensing can be defined as the measurement of objects of interests' properties on the Earth surface using data acquired from aircraft and satellites. Remote sensing systems, especially satellite-based systems, provide many useful applications due to their ability to provide a repetitive and consistent view of the Earth. These applications include environmental assessment and monitoring, natural resource exploration, military surveillance and topography mapping.[1]

Due to the airborne and spaceborne nature of remote sensing systems, the systems rely on the propagated signals, such as electromagnetic waves, for information acquisition. The acquisition is done by specialized hyperspectral sensors that produce a 3-D hyperspectral image that consists of a few hundred spectral bands across the visible and infrared regions of the electromagnetic spectrum in one run with high spectral resolution. This produces very large hyperspectral images.

This also provides a constant challenge for spaceborne remote sensing systems, which require the acquired image to be downloaded by operators. This is because the satellites have limited onboard storage and downlink bandwidth.

To overcome this limitation, the hyperspectral image data undergo compression just after the acquisition occurs. Lossless compression is preferred for satellite-borne applications due to the need for post-processing at high fidelity and resolution.[2] As a response, the Consultative Committee for Space Data Systems (CCSDS) has released the CCSDS 123.0-B-1 Lossless Multispectral and Hyperspectral Image Compression (CCSDS-MHC) standard as the main standard for lossless hyperspectral image compression.[3]

While the CCSDS-MHC algorithm provides state-of-the-art compression performance in low-complexity domains, there is always some need to improve various aspects of performance, such as execution time.[4] One solution is to undergo parallelization.

Parallelization involves dividing a programme into several parts that can be solved at the same time with the same result. There is increasing interest in introducing multicore processors for next-generation satellites that could make use of parallelization for various tasks on satellites.[5]

To apply parallelization in the CCSDS-MHC algorithm, APIs such as OpenMP can be used. OpenMP consists of a set of compiler directives and a library of support functions in C/C++ for programme parallelization. By finding the right section of the CCSDS-MHC algorithm to be parallelized, the execution time of the algorithm can be greatly decreased.

2. ALGORITHM OVERVIEW

The CCSDS-MHC algorithm is a variation of the fast lossless (FL) compression algorithm that uses 3D compression techniques.[6] In the algorithm, only the compression part is specified and standardized.[3] Nevertheless, both the compression and decompression parts of the algorithm will be introduced.

Fig. 1 shows the components of the compressor. The compressor of the CCSDS-MHC algorithm consists of two parts: the predictor and encoder. The predictor predicts the value of each image sample based on the values of a nearby sample in a small 3D neighborhood. The main difference between the predicted and actual sample values, or prediction residual, is the output of the predictor.

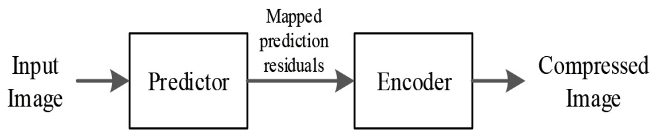

Figure 1: Compressor Schematic

Fig. 2 shows the flowchart of the predictor process based on steps specified by the standard.[3] The encoder then encodes the mapped prediction residuals based on a statistical method known as the sample-adaptive entropy-coding approach. This forms sequences of data bits that consist of a header followed by the body. The header contains the compression parameters of the process, while the body contains codewords that are coded as mapped prediction residuals. The lengths of codewords vary according to the occurrence of the residual's value. A value with a higher occurrence will have shorter codewords. The data bits are then written to an external file bit by bit and become the compressed image.

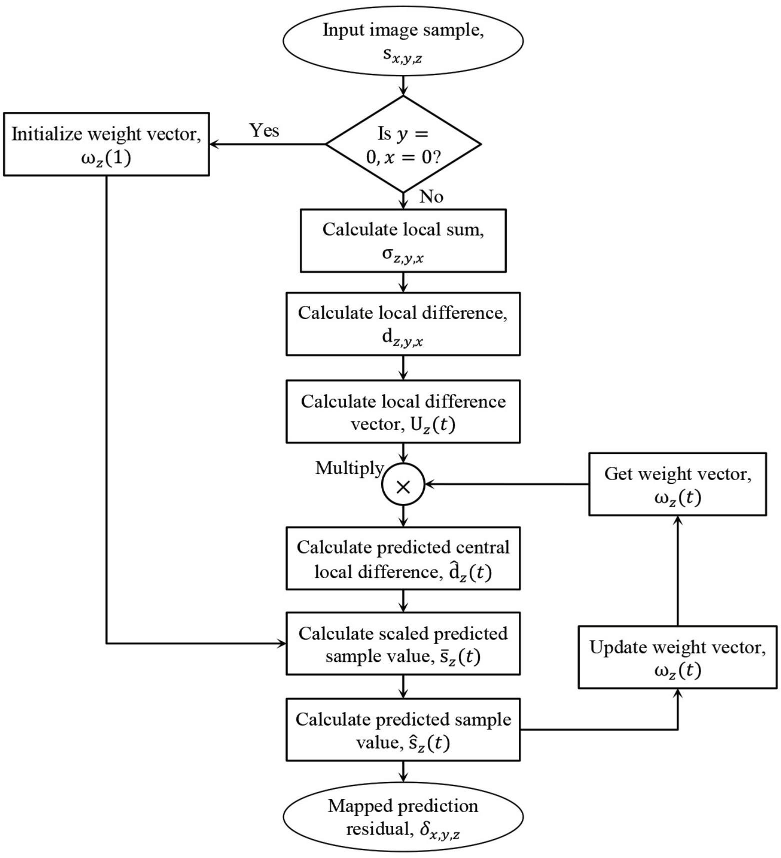
Input image sample, $s_{x,y,z}$
Is $y = 0, x = 0$?
Yes
No
Initialize weight vector, $\omega_z(1)$
Calculate local sum, $\sigma_{z,y,x}$
Calculate local difference, $d_{z,y,x}$
Calculate local difference vector, $U_z(t)$
Multiply
$\times$
Get weight vector, $\omega_z(t)$
Calculate predicted central local difference, $\hat{d}_z(t)$
Calculate scaled predicted sample value, $\bar{s}_z(t)$
Update weight vector, $\omega_z(t)$
Calculate predicted sample value, $\hat{s}_z(t)$
Mapped prediction residual, $\delta_{x,y,z}$

Figure 2: Flowchart of the Prediction Process

The decompressor is the inverse of the compressor. Fig. 3 shows the components of the decompressor. Similarly, the decompressor also consists of two parts: a decoder and an inverse predictor. Both parts are the direct inverse of the compressor's encoder and predictor, respectively. The decoder converts codewords back into mapped prediction residuals, while the inverse predictor converts the residuals back to the original image sample value. Both functions are similar to their counterparts, as the function still uses the same prediction method and statistical method, albeit with slightly different formulas for some parts of the calculations.

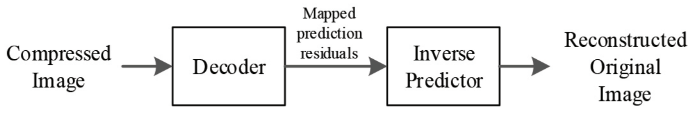

Figure 3: Decompressor Schematic

3. PREVIOUS WORKS ON CCSDS-MHC PARALLELIZATION

Several literature reviews of previous works on CCSDS-MHC parallelization have been conducted to further understand the method of parallelization. A summary of these works is as follows.

Hopson et al. analyse the CCSDS-MHC algorithm and identify the part where parallelization can be applied with the usage of multicore central processing unit (CPU) architectures.[7] In the paper, OpenMP is used for parallelization for a fixed platform (desktop) and mobile platform (laptop). The parallelization approach was used to buffer the data whenever there is a dependency on the data. This prevents the situation where one parallel execution thread needs to wait for data since the algorithm stage only begins when all the data required are computed. In fixed platforms, it was reported that parallelization can be efficiently done during the stages involving input formatting, data encoding and data packing. However, for the predictor stage, there was the need for a very large buffer, and it was not feasible for the parallelization process. However, under mobile platforms using Mobile OpenMP, the introduction of a buffer was not required. Instead, the image segmentation method is used. The input image was split into smaller parts, and each part underwent compression in a serial manner, albeit in different cores of the processor. The execution time was significantly faster.

Davidson and Bridges used CUDA, another parallel computing API, to perform CCSDS-MHC compression in GPU.[4] The team used a multispectral image as an input, which contains a smaller number of bands (less than 10) compared to hyperspectral images. The image was segmented into individual tiles and compressed individually. Then, the team modified the CUDA programs to allow them to compress multiple tiles within a single thread block. With this implementation, a speed-up of 2.41 times was achieved. However, the team also concluded that this method was not so convincing when hyperspectral images are used as input.

Schwartz and Pinho used an embedded multicore platform to test several common image compression algorithms, including CCSDS-MHC.[8] Input images were segmented into tiles and compressed individually. In this study, the team tried to find the effect of the number of images segmented on the execution time of the algorithms running in parallel. The team revealed that while separating the image into multiple tiles certainly makes the algorithms run faster. The tiles splitting can also increases the execution time compared to a low number of tiles. This is because the platform needs to expend time managing many tasks.

Olaru and Craus suggested an effective parallelization scheme for CCSDS-MHC.[9] The 3D hyperspectral image was first decomposed into relatively independent blocks. These blocks were then distributed by a data manager into all available processing cores. A scheme known as the master-worker design paradigm was used. It consists of a single master node and multiple worker nodes. The master node splits the work into workers and then becomes one of the workers. However, it also coordinates the worker and collects the results from the worker. When a worker node finishes its assigned work, it receives the next task. This happens until all tasks are executed. One of the notable challenges arising in this method is the need for the extra effort in finding a way to split up the mapped prediction residuals into blocks for the encoding process.

Rodríguez et al. put spatial clustering (tiling), also known as segmented images, into the CCSDS-MHC compressor on a commercially available System on Programmable Chip (SoPC), ARTICo^3 architecture. AVIRIS images with a spatial size of 512512 are tiled spatially with various sizes of up to 88 pixels. However, tiling is known to affect CR performance in comparison to the case without tiling, but it is beneficial to fault tolerance. Each sub-image (called a segment) would introduce its header, which makes the CR performance worse. Each segment is processed independently in an accelerator to speed up performance. The system managed to reach a throughput of 67 MSamples/s by using 16 accelerators.[10]

With the previous studies on improving the speed/throughput for the CCSDS-MHC algorithm using various hardware, such as multicore CPUs and GPUs, it is necessary to investigate the algorithm and optimise the flow so that the parallelization is performed efficiently without any buffer memory. In the algorithm, the local sum, $\sigma_{z,y,x}$ (shown in Fig. 2), introduces data dependency that requires buffer memory and prevents parallelization on that particular part of the algorithm. This study overcomes this bottleneck to provide better parallelization to the algorithm. It is expected that the enhancement made in this research can be implemented on a hardware platform, such as a multicore processor, which is more suitable to use on satellite platforms than a desktop computer or GPU. This study focused on the optimum parameters for the CCSDS-MHC algorithm to produce the best compression ratio (CR) without tiling for lossless hyperspectral images by utilising the full prediction mode of the standard, as suggested by Sanchez, Auge.[11]

4. METHODOLOGY

The development of parallelized CCSDS-MHC involves the four steps described in Fig. 4. Since the OpenMP is available in the C/C++ language, the CCSDS-MHC algorithm first needs to be coded in C/C++ based on the Java implementation available from the Group on Interactive Coding of Images (GICI).[12] The algorithm will then be tested to ensure it works correctly. Next, the algorithm is parallelised by using OpenMP before testing the algorithm to assess its performance.

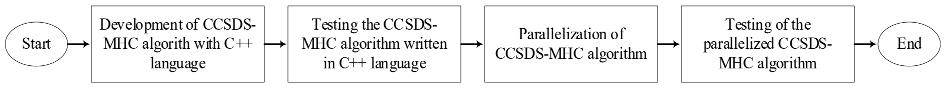
Start
Development of CCSDS-MHC algorith with C++ language
Testing the CCSDS-MHC algorithm written in C++ language
Parallelization of CCSDS-MHC algorithm
Testing of the parallelized CCSDS-MHC algorithm
End

Figure 4: Flowchart of the Development of the Parallelised CCSDS-MHC Algorithm

4.1 C/C++ Coding of the CCSDS-MHC Algorithm

The main objective of this research was to create a parallel version of the CCSDS-MHC algorithm, therefore the algorithm was simplified considerably. First, in the original CCSDS-MHC algorithm, there are choices for the user to specify the method of calculating local sums, the prediction mode and the type of encoding approach. However, in this work, these choices are restricted to the neighbour-oriented method, full prediction mode and sample-adaptive entropy coding approach fixed throughout the research. These configurations are chosen for the best lossless CR performance.[3] Second, each user-defined parameter throughout the CCSDS-MHC algorithm is fixed to an optimum value suggested by Sanchez, Auge.[11] Since the objective of this research is to evaluate the CCSDS-MHC algorithm in terms of execution time, these parameters are fixed to an optimum value so that the best CR can be obtained. The suggested parameters are listed in Table I below.

Table I: Parameters Suggested for the CCSDS-MHC Algorithm11

Parameter	Alias	Value
Previous bands, P	NUMBER_PREDICTION_BANDS	15
Dynamic range, D	DYNAMIC_RANGE	16
Output word size	OUTPUT_WORD_SIZE	4
Predictor meta-data flag	PREDICTOR_METADATA_FLAG	False (0)
Coder meta-data flag	ENTROPYCODER_METADATA_FLAG	False (0)
Local sum	LOCAL_SUM_MODE	Neighbour oriented mode (0)
Prediction mode	PREDICTION_MODE	Full prediction mode (0)
Register size	REGISTER_SIZE	32
Weight initialisation method	WEIGHT_INITIALIZATION_METHOD	Default (0)
Weight initialisation table flag	WEIGHT_INITIALIZATION_TF	False (0)
Weight component resolution	WEIGHT_COMPONENT_RESOLUTION	13
Weight update scaling exponent change interval	WEIGHT_UPDATE_SECI	64
Weight update scaling exponent initial parameter	WEIGHT_UPDATE_SE	-1
Weight update scaling exponent	WEIGHT_UPDATE_SEFP	3
Adaptive encoder	ENTROPY_CODER_TYPE	Sample adaptive encoder (0)
Unary length limit	UNARY_LENGTH_LIMIT	16
Rescaling counter size	RESCALING_COUNTER_SIZE	6
Initial count exponent	INITIAL_COUNT_EXPONENT	1
Initialisation accumulator table flag	ACCUMULATOR_INITIALIZATION_TF	False (0)
Accumulator initialisation constant	ACCUMULATOR_INITIALIZATION_ CONSTANT	5

Both the compression and decompression parts of the algorithm are coded in C/C++ in this research. This provides an easy way to test the validity of the coded algorithm by joining both parts together, which makes the output of the compressor the input of the decompressor. The correct algorithm will make its original image (input of the compressor) and its decompressed image (output of the decompressor) have the same value, pixel by pixel since the algorithm is lossless. The act of joining each part requires sharing an image array that consists of codewords. This requires additional care to eliminate the leading zeros problem of the codewords, where different codewords will have the same value when converted into integers. For example, codewords ‘01’ and ‘001’ will have the same values.

4.2 Parallelization of the CCSDS-MHCc Compression Algorithm

In this research, only the compression part of the CCSDS-MHC algorithm will be parallelised. The first step of the parallelization is to identify the parts of the algorithm that are parallelizable. This can be done by drawing a data dependence graph for the algorithm, as shown in Fig. 5.

In Fig. 5, each band of the image has a large potential to be parallelized. The compression process for one band does not depend on the compression processes of another band. In the original algorithm, the compression process starts from the first band and continues to the last band, one at a time. This shows that the algorithm exhibits data parallelism for each band. Nevertheless, there is one data dependence present in the algorithm: the process of calculating the local sum, which involves the generation of a prediction neighbourhood. The generation of a prediction neighbourhood requires all the sample values of the current and a certain number of previous bands. However, since all pixel values of all bands are already obtained when the image is loaded, this problem can be solved by requesting the values of the involved bands required directly from the loaded image at the start of compression for every core, as opposed to the original algorithm, where band shifting is conducted.

The parallelization of the algorithm was performed using the OpenMP’s parallel for schedule directive. An ordinary parallel for a directive, which is common for coding data parallelism, is not used since the iterations are conducted by a random processor, and this requires the sample value requested for neighbourhood generation to be conducted for every band, which is slower compared to the band shifting. Under the parallel for schedule directive, a contiguous range of iterations called chunks is assigned to each processor. For example, when compressing 8-band images under a system with two processors, a processor will compress bands 0 to 3 and another processor will compress bands 4 to 7. This enables the sample value requesting method to run only one time, followed by a band shifting method for subsequent bands, which saves a lot of time.

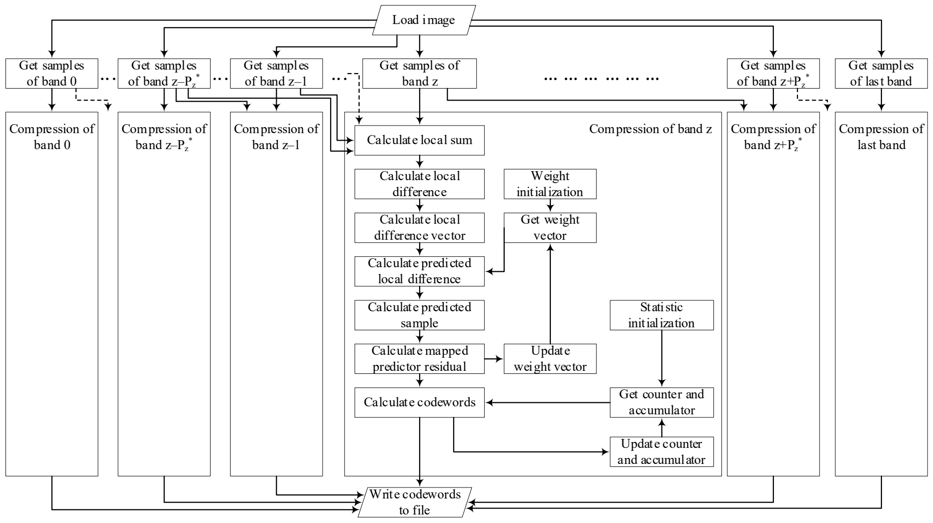

Figure 5: Data dependence graph of the CCSDS-MHC compression algorithm

4.3 Testing Images and Environment

Four Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) images, consisting of 224 spectral bands, and four Hyperion images were used for testing purposes.[13, 14] All the AVIRIS and Hyperion images were cropped from the top-left pixel to form a 512512 and 256256 spatial size, respectively. Only 196 spectral bands, which were classified as unique and calibrated by Thenkabail et al., were extracted and used for the Hyperion images.[15] The images are listed in Table II.

Table II: AVIRIS and Hyperion Test Images

Dataset	Hyperspectral Image (Abbrev.)	Image Size (RowColumnBand)
AVIRIS	Cuprite Scene 1 (Cuprite1)	512512224
	Jasper Ridge Scene 2 (Jasper2)	
	Low Latitude Scene 1 (Low1)	
	Lunar Lake Scene 2 (Lunar2)	
Hyperion	EO1H0120312001129111P1_PF1_01 (Boston)	256256196
	EO1H0140362001127110PP_AGS_01 (Edenton)	
	EO1H0090112001140111PP_PF1_01 (Greenland)	
	EO1H0150332001134111P1_AGS_01 (Portobago)	

Next, three computer systems with different numbers of processors (threads) were used to evaluate the parallelised CSSDS-MHC algorithm. The specifications of the systems are shown in Table III.

Table III: Computer System Used for Algorithm Evaluation

	4-Thread System	8-Thread System	16-Thread System
OS	Windows 10, 64-bit		
CPU Type	Intel Core i7-7500U (mobile)	Intel Core i7-6700K (desktop)	Intel Xeon E5-1660 v4 (server)
Clock Speed	2.7 GHz	4.00 GHz	3.2 GHz
CPU cores	2	4	8
CPU threads	4 threads	8 threads	16 threads
RAM	12GB	8 GB	16 GB
Storage Type	HDD	HDD	HDD
L1 Cache	128 KB	256 KB	512 KB
L2 Cache	512 KB	1.0 MB	2.0 MB
L3 Cache	4.0 MB	8.0 MB	20.0 MB

4.4 Performance Indicator of the Parallelised Algorithm

The main performance indicator to assess the parallelised CCSDS-MHC algorithm is execution time. The execution time of a CCSDS-MHC algorithm starts at the time when the input image is done loading and ends when the last pixel is compressed and its codewords are written to the file. The improvement in execution time between sequential and parallel versions of the algorithm is quantified by speedup. Speedup is defined as the ratio between sequential execution time and parallel execution time.[16]

Another common performance indicator, the compression ratio, is also calculated, although it is not important in this research. This is because the compression ratio is always the same with the same image since all the user parameters are fixed. The calculation is performed simply for reference purposes.

5. RESULTS AND DISCUSSION

5.1 Compression Ratio of the CCSDS-MHC Algorithm

Table IV shows the compression ratio (CR) of the images tested with the algorithm.

Table IV: Compression Ratio of Tested Images

Hyperspectral Image	Original Size (MBytes)	Compressed Size (MBytes)	CR
Cuprite1	117.44	35.81	3.28
Jasper2		36.05	3.26
Low1		39.29	2.99
Lunar2		36.13	3.25
Boston	25.69	12.83	2.00
Edenton		12.85	2.00
Greenland		10.93	2.35
Portobago		10.39	2.47

The table shows that there seems to be some correlation between the size of the images and CR. Such a correlation implies that the compression of smaller images results in less compression. A plausible explanation for this correlation is that the average bit length of the codewords is nearly the same for most of the images. This causes the rate of decrease of uncompressed image sizes to be faster than the rate of decrease of compressed image sizes, in turn causing a decrease in CR.

5.2 Speedup of the Parallelised CCSDS-MHC Algorithm

Table V shows the average execution times for each dataset in each implementation. Table VI shows the speedups of all systems and the theoretical maximum speedup according to the Amdahl's Law, shown in (1), where is the number of processor threads in a system and is the fraction of the operations in a computation that must be performed sequentially. In this research, the value of is fixed to 0.1 (i.e., 10%) through rough estimation from the data dependency in the local sum calculation. The maximum speedup by the Amdahl's Law is provided as a reference only.

Maximum speed up, $\psi = \dfrac{1}{f + \frac{1}{P}(1 - f)}$

Table V: Average Execution Time (seconds) for Sequential and Parallel Implementation

	Execution Time (seconds)					
	4-Thread System		8-Thread System		16-Thread System	
Dataset	Sequential	Parallel	Sequential	Parallel	Sequential	Parallel
AVIRIS	40.40	17.99	26.64	10.66	21.55	4.02
Hyperion	5.86	3.53	2.58	1.58	3.99	0.57

Table VI: Speedup of Tested Datasets for Each System

	Speedup		
Dataset	4-Thread System	8-Thread System	16-Thread System
AVIRIS	2.25	2.50	5.37
Hyperion	1.66	1.63	7.06
ψ with $f = 0.1$	3.08	4.71	6.40

Table VI shows that there is a significant speedup between the sequential and parallel versions of the algorithm across the tested systems for each hyperspectral dataset. For all the systems and datasets, a speedup value larger than one shows that the parallel implementation is faster than its corresponding sequential implementation. The 4-thread system and 8-thread system recorded almost the same performance in terms of the speedup value, while the 16-thread system had a far higher speedup value. Even though there was an almost identical speedup value between the 4- and 8-thread systems, the execution time (Table V) still favoured the 8-thread system due to differences in architecture (the 4-thread system was mobile and the 8-thread system was desktop), higher clock speed and bigger cache sizes. The larger RAM size in the 4-thread system in comparison to the 8-thread system does not affect the results. For the 16-thread system, which is a server-grade computing platform, similar trends have been observed, except for the processor's clock speed, which is slightly lower than that of the 8-thread system.

In reference to the maximum speedup, ψ by Amdahl's Law, the differences between the measured results and the theoretical speedup are because the systems used in this research have different specifications, such as processor's thread count, cache sizes, processor speeds and architectures, as well as RAM sizes. The processor's thread count is the main factor to get the best execution time (in addition to the other factors, such as cache size, clock speed and RAM size), due to the parallelization of the algorithm that has been performed based on the available processor's thread, as explained before. The CR from all systems is similar, as shown previously in Table IV.

When comparing the datasets, Hyperion can be executed faster than AVIRIS (Table V), mainly due to the smaller size (both spatially and spectrally) in the former dataset. This can also be observed in Table VI in terms of speedup. Moreover, the speedup for the Hyperion dataset in the 16-thread system (7.06) is larger than its theoretical speedup (6.40). Since OpenMP automatically handles the parallelization process between the processor thread, the value of $f = 0.1$ might not be suitable for the system and causes the difference.

6. CONCLUSION AND FUTURE WORK

In conclusion, a hyperspectral image compression system based on the CCSDS-MHC algorithm with the support of parallel processing has been successfully designed and developed. The algorithm is divided into chunks of bands and runs concurrently. This parallelization is done using OpenMP's parallel for schedule directives. The parallelized algorithm is indeed run at a higher speed compared to its sequential counterparts, with a speedup of about 1.6 times to 7.0 times, depending on the number of threads in the system running the algorithm and the type of images. The AVIRIS dataset can be compressed from around 4 to 18 seconds in the parallel implementation, compared to 21 to 40 seconds in the sequential counterparts. The Hyperion dataset, which has a smaller size, can be compressed from under one second to around four seconds in parallel compared to four to six seconds in the sequential implementation. Unlike the other studies that focused on tiling, this study focuses on the full spatial resolution of the images, which produces the best CR performance. This study also uses the full prediction mode to achieve the same objective.

In future research, it would be worthwhile to explore the impact of the parallelized algorithm in terms of compression ratio and execution time when user-defined parameters are changed. The implementation of hardware (multicore processors such as DSP and FPGA) is expected to produce a faster execution.

7. ACKNOWLEDGEMENTS

This work was supported by USM Short Term Grant 304/PELECT/6315003.

8. REFERENCES

1. Schowengerdt, R.A., *Remote Sensing: Models and Methods for Image Processing*. 3rd. ed. 2012: Elsevier Inc. 515.
2. Lopez, G., E. Napoli, and A.G.M. Strollo. *FPGA implementation of the CCSDS-123.0-B-1 lossless Hyperspectral Image compression algorithm prediction stage*. in *2015 IEEE 6th Latin American Symposium on Circuits & Systems (LASCAS)*. 2015. DOI: 10.1109/LASCAS.2015.7250438.
3. CCSDS, *Lossless Multispectral & Hyperspectral Image Compression - Recommended Standard (CCSDS 123.0-B-1)*. 2012, Washington DC, USA: CCSDS Secretariat, NASA. 52.

4. Davidson, R.L. and C.P. Bridges. *GPU accelerated multispectral EO imagery optimised CCSDS-123 lossless compression implementation*. in *2017 IEEE Aerospace Conference*. 2017. DOI: 10.1109/AERO.2017.7943817.
5. Andersson, J., J. Gaisler, and R. Weigand. *Next Generation MultiPurpose Microprocessor*. in *DASIA 2010 - Data Systems In Aerospace*. 2010. DOI: 2010ESASP.682E...8A.
6. Matthew, K. *Low-complexity adaptive lossless compression of hyperspectral imagery*. in *Proc.SPIE*. 2006. DOI: 10.1117/12.682624.
7. Hopson, B., et al. *Real-time CCSDS Lossless Adaptive Hyperspectral Image Compression on Parallel GPGPU & Multicore Processor Systems*. in *NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2012*. 2012. Erlangen, Germany DOI: 10.1109/ahs.2012.6268637.
8. Schwartz, C. and M.d.S. Pinho, *Remote-Sensing Image Compression Using Embedded Multicore Platforms With Energy Consumption Measurement.* IEEE Geoscience and Remote Sensing Letters, 2015. **12**(12): p. 2453-2457 DOI: 10.1109/LGRS.2015.2484076.
9. Olaru, M. and M. Craus. *Lossless multispectral and hyperspectral image compression on multicore systems*. in *2017 21st International Conference on System Theory, Control and Computing (ICSTCC)*. 2017. DOI: 10.1109/ICSTCC.2017.8107030.
10. Rodríguez, A., et al., *Scalable Hardware-Based On-Board Processing for Run-Time Adaptive Lossless Hyperspectral Compression.* IEEE Access, 2019. **7**: p. 10644-10652 DOI: 10.1109/ACCESS.2019.2892308.
11. Sanchez, J.E., et al. *Review and Implementation of the Emerging CCSDS Recommended Standard for Multispectral and Hyperspectral Lossless Image Coding*. in *2011 First International Conference on Data Compression, Communications and Processing*. 2011. DOI: 10.1109/CCP.2011.17.
12. GICI. *Emporda Software*. 2021 9 December 2021]; Available from: http://gici.uab.es/GiciWebPage/downloads.php#emporda.
13. NASA-JPL. *Ordering Free AVIRIS Standard Data Products*. 2015 9 December 2021]; Available from: https://aviris.jpl.nasa.gov/data/free_data.html.
14. EROS. *USGS EROS Archive - Earth Observing One (EO-1) - Hyperion*. 2019 9 December 2021]; Available from: https://www.usgs.gov/centers/eros/science/usgs-eros-archive-earth-observing-one-eo-1-hyperion.

15. Thenkabail, P.S., et al., *Hyperion, IKONOS, ALI, and ETM+ sensors in the study of African rainforests.* Remote Sensing of Environment, 2004. **90**(1): p. 23-43 DOI: https://doi.org/10.1016/j.rse.2003.11.018.
16. Asanovic, K., et al., *A view of the parallel computing landscape.* Commun. ACM, 2009. **52**(10): p. 56–67 DOI: 10.1145/1562764.1562783.