

# Real-time Adaptive Lossless Hyperspectral Image Compression using CCSDS on Parallel GPGPU & Multicore Processor Systems

Ben Hopson, Khaled Benkrid  
School of Engineering and Electronics  
The University of Edinburgh  
Edinburgh, UK  
B.Hopson@ed.ac.uk

Didier Keymeulen, Nazeeh Aranki, Matt Klimesh,  
Aaron Kiely  
Jet Propulsion Laboratory  
California Institute of Technology  
4800 Oak Grove Dr., Pasadena, CA 91109, USA  
didier.keymeulen@jpl.nasa.gov

**Abstract**— The proposed CCSDS (Consultative Committee for Space Data Systems) Lossless Hyperspectral Image Compression Algorithm was designed to facilitate a fast hardware implementation. This paper analyses that algorithm with regard to available parallelism and describes fast parallel implementations in software for GPGPU and Multicore CPU architectures. We show that careful software implementation, using hardware acceleration in the form of GPGPUs or even just multicore processors, can exceed the performance of existing hardware and software implementations by up to 11x and break the real-time barrier for the first time for a typical test application.<sup>1</sup>

**Keywords**—CCSDS; Lossless; Hyperspectral; Image Compression; Adaptive Filter; Multicore; GPGPU; Parallel; Realtime

## I. INTRODUCTION

Hyperspectral image sensors may be mounted on satellites or high-altitude aircraft, and used for land-survey, mineral prospecting, and reconnaissance. They are now seeing increased use on unmanned aerial vehicles (UAVs) for military reconnaissance due to their ability to identify camouflage, based on spectral characteristics. In the space-exploration community, there is great interest in adding hyperspectral sensors to all missions, as they perform the work of many sensors / spectrometers.

While space application require special radiation-hard hardware like the Xilinx Virtex 5QV FPGA, UAVs and high-altitude airframes may employ standard PC technology, which could include general purpose graphics processors (GPGPUs). Hyperspectral sensors generate a very large amount of data, and down-link bandwidth is a scarce resource on all platforms, so there is great interest in employing compression to maximise this bandwidth. There is also utility in employing compression for images at the archival stage, on the ground. This paper describes the implementation of a well-tested algorithm (CCSDS Lossless) on GPGPU accelerated and

multicore PC architectures, suitable for integration with data collection devices on an aerial platform and for ground use.

Visible light image sensors use sensor elements tuned to 3 colour bands (red, green & blue) but hyperspectral sensors use a diffraction grating to split light into its spectral components and scan a high sensitivity sensor over the desired range, giving an output vector with typically several hundred band measurements. Each spatial image pixel consists of this long spectral amplitude vector.

Sensor data are traditionally presented in spectral band major order (Band Interleaved Pixel format, or BIP). All spectral bands for a single image pixel are emitted before the sensor moves to the next spatial pixel. This is the standard output format for so called whisk-broom type sensors.

Newer sensors (particularly the so called push-broom type) are starting to output data in line-major order (Band Interleaved by Line format, or BIL) – where a whole image line (or part of an image line) is output from one spectral band, before any data are produced from the next spectral band.

Push-broom sensors collect light from many spatial pixels simultaneously. Consequently, they have higher data output rate, and also tend to have higher sensitivity, as the sensor dwells on each pixel for longer. For a comprehensive review of remote sensing technologies, see [1]

As with normal images, spatially close pixels are highly correlated. Hyperspectral images also exhibit high correlation of adjacent spectral bands. The challenge in designing a compression algorithm for hyperspectral data is to make use of both the spectral and spatial correlation properties. The challenge for implementing such an algorithm is to cope with the high data rate produced by such sensors.

An initial JPL project focussed on a hardware implementation for space hardened Virtex 4 FPGA technology. This was optimised for processing streaming data coming directly from a sensor. Although heavily pipelined, this algorithm implementation is essentially serial, see [2]

In a joint project between JPL and Edinburgh University, two new implementations were developed, designed to run in software on off-the-shelf PCs. One implementation makes use

---

<sup>1</sup> We define ‘real-time performance’ to mean: having throughput at least 800Mb/s to match a typical sensor.

of GPGPU acceleration, the other used multicore CPU. The joint development [3] was the first truly parallel implementation of the CCSDS Lossless algorithm. Although exhibiting a huge speedup compared to serial software implementations and even beating the pure hardware implementation – that parallel software version did not reach real-time performance for the test application.

The original goal for these new implementations was to see how much of the performance could be transferred to a mobile (laptop) platform, which could be moved easily between an airframe and ground stations. The architectural improvements made proved to be so significant, however, that the performance of this new implementation exceeds that of all existing versions by a factor 11 when running on comparable hardware, and comprehensively breaks the real-time barrier – even without GPGPU acceleration.

Despite being designed for mobile computing, there are no specific features which require these implementations to be run on a mobile device. Equivalent or better performance can be obtained from this implementation when run on a standard high performance workstation. The laptop used for development and testing was a high performance gaming grade device, equipped with a pair of Nvidia 500 series mobile chipset graphics cards. See TABLE III. This represents the current highest performing available mobile hardware, however significantly more computing power is available on workstation platforms since they have fewer limitations on power consumption and thermal envelope, not to mention weight.

A key difference between the software implementations described here and the hardware implementation in [2] is that the hardware version was optimised for processing incoming data presented as a single serial stream, while the software versions described here are designed to operate on complete images or sections of images, presented as complete files stored on disk. We shall describe how this extra stage of buffering is extremely useful in providing an extra axis for parallelisation.

This paper will focus on the architectural and performance differences between the various software implementations, with and without GPGPU acceleration.

The remainder of this paper is organised as follow:

Section II contains a summary of the algorithm itself, with particular reference to the sources of parallelism, and impediments to parallel implementation.

Section III provides an overview to the various implementations, and contrasts these with the existing Fixed Platform versions.

Section IV describes the implementation of a software decompressor, which is a new feature added in the Mobile Platform implementations.

Section V lists the features of the specific hardware and testing methodologies used to derive the results presented in this paper.

Section VI contains the performance results, comparing the performance of the new Mobile Platform implementations with their Fixed Platform counterparts.

Finally, Section VII describes directions for further work and development of the ideas presented in this paper.

## II. ALGORITHM OVERVIEW

The CCSDS Lossless Hyperspectral Image Compression algorithm is well described in the original FPGA implementation paper [2], as well as the more recent software implementation paper [3], and the specification itself [4].

The key design goals for the algorithm itself were that it should maximise compression performance, but minimise the amount of logic required when implemented in hardware. For this reason, the reduced predictor needs only 3 integer multiplies<sup>2</sup> to process each input sample, with all other operations implemented as shifts and add operations. The existing hardware implementations are heavily pipelined for speed, which is a form of data parallelism. Software cannot directly use the same form of parallelism – so we instead look to various sorts of block-scale data parallelism.

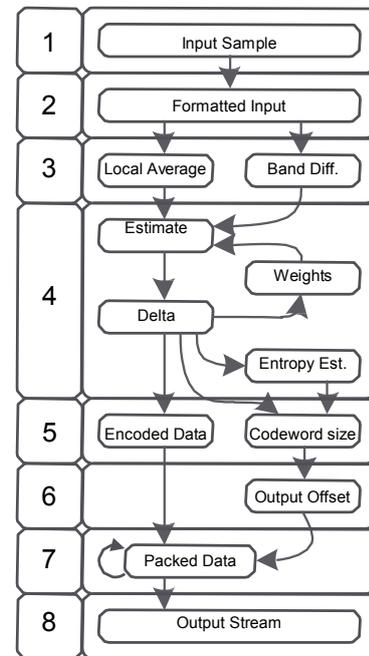


Figure 1 - Data Dependency Graph for the Lossless CCSDS Compression Algorithm

The algorithm was not designed to have a high degree of block-parallelism, and various data-dependencies imply a serial implementation. See Figure 1. We shall see that, with care and suitable buffering, most of these serial dependencies can be removed. It is useful to summarise here these data-

<sup>2</sup> The standard allows for any number  $P_z$  of bands to be used in reduced mode. The full predictor then uses  $P_z+3$  bands, and hence requires this many multipliers as well.

dependencies within the algorithm since these have major implications for the implementations described below.

#### A. Input Reformatting

For good performance, it is vital that data are presented to the algorithm in the order they are to be used. If the data are ordered BIL, we must first reorder to BIP format. If the data are big-endian and we are running on a little-endian architecture, we must also byte reverse each input sample.

The BIP/BIL conversion can be a difficult to perform efficiently on GPGPU; the operation is essentially matrix transposition. The approach that our implementation takes is to transpose small sub-matrices held in buffers, and the sizes of these buffers are chosen at run-time based on the available hardware and problem size.

At this stage, there are no dependencies between samples and all operations could potentially occur in parallel.

#### B. Fixed Filter

The multiple sensors used in push-broom devices will exhibit differences in sensitivity. Vertical stripes of pixels will be scanned by the same sensor, and pixels will exhibit stronger correlation in the vertical than the horizontal axes. Whisk-broom devices do not have this issue, since the same sensor is used to detect all pixels. To exploit this anisotropy, there are two versions of the compression algorithm. One is optimised for whisk-broom and uses a 2D fixed spatial filter first-stage ('Neighbour Average'), and the other is optimised for push-broom and uses a 1D vertical filter first-stage ('Column Average'). This paper focusses on the push-broom optimised version, also called the reduced predictor algorithm.

The fixed filter aims to remove as much structural spatial correlation from the data as possible before reaching the adaptive filter. In the reduced predictor mode, this is simply a 1D high-pass filter, aligned with the vertical axis – that is, samples are differenced against their vertical neighbour, in each spectral band independently.

The outputs of this convolution are independent of each other spectrally and spatially, and so the convolution could potentially be performed on the entire data-cube in parallel, if we had enough computing resources available.

In the streaming hardware implementation, this requires a large buffer to store a complete image line (in all spectra). In software, we are operating on complete images, so this requirement is met automatically.

The band difference vector for each spectral band shares all but one value with the band difference vector from the previous band. In hardware this is easily implemented as a shift-register. In software we can share pointers between bands, but this introduces a dependency. We need to make sure that all of the individual band-differences for the spectral bands of one spatial pixel are calculated before we try and reference the vector itself; otherwise we could end up referencing old data. This is the same as saying that we require a spectral axis barrier synchronisation following the calculation of the band-difference vector for a spatial pixel.

#### C. Adaptive Filter

After removing structural spatial correlation with a fixed filter, the standard has an adaptive filter as a predictor. Encoder and decoder devices will derive identical predict values, and so all that needs to be sent is the departure of the data from the prediction (the predictor error). The feedback in the adaptive filter works to minimise the predictor error.

There are many approaches to adaptive filter design, but the one used in the standard is to just take the sign of the error in the feedback loop. A scheduler controls the feedback gain, so we can have both fast lock-on at the start and good steady-state performance.

The adaptive filter is presented with spatial pixels in raster-scan order, and updates its weights after each pixel. This puts a hard serial dependency between pixels in the spatial axis.

Due to the clipping operation that occurs in the weight update and estimate calculations, operations in the spatial axis do not commute. If we were talking about stochastic processes, we would say that the weight update process is path-dependent.

Although this forces the predictor to run serially in the spatial dimension, the only interdependence between spectral bands comes through the band-differences vector, which we have already effectively decoupled. Therefore, we can still run the predictor band-parallel, that is, in parallel across the spectral bands of a pixel.

Ultimately, this property imposes the tightest restriction to parallelism for the whole algorithm, as the predictor forms both the critical path (it contains the only expensive operations, the multipliers) and the parallelism bottleneck. It is not possible to avoid this without fundamentally changing the algorithm, but we shall see that it is possible to mitigate it by developing parallelism from other sources.

#### D. Adaptive Encoder

The error signal from the adaptive filter has very much lower amplitude than the original signal, and ideally looks like the noise component of our image compared with an idealised hyperspectral image model.

As filtered noise, the signal should exhibit a 2-sided exponential distribution. The standard therefore folds the negative half into the positive half to get a 1-sided exponentially distributed random variable, and encodes this optimally with a Golomb-Rice code. For the original paper on such codes, see [5] or [6] for an example of a more modern application.

To simplify implementation, the standard limits itself to power-of-two code sizes ( $2^k$ ). As an extra level of adaptation, the code-word parameter  $k$  is allowed to vary, tracking the average entropy of our signal, using the notation of [7]. Separate spectral bands are treated as different contexts for the purposes of our code.

In other words, the average amplitude of the output error signal is tracked *within each spectral band*, and finds  $k(z)$  for each band such that  $2^{k(z)}$  bounds the average magnitude of the error, as described in detail in [7].

Since the average magnitude of the error within each band is needed at each spatial pixel – this imposes a spatial serial restriction to the encoder, just like that in the predictor.

We can do slightly better than this at implementation, however, by separating the entropy estimation operation (which needs to be pixel-serial) from the actual encoding operation (which is band and pixel independent once the average entropy is known). The encoding operation consists of masking off the bottom  $k$  bits of the error (suitably sign encoded), and unary-encoding the remaining high bits. All this can be done with bit-operations and shifts. Provided  $k$  is known for each error sample, this encoding can be performed in parallel across all bands and across all pixels.

### E. Output Serialiser

The final part of the algorithm is an operation which is trivial in hardware, but expensive in software. The output from the adaptive encoder is a series of variable length code-words. These need to be output serially (in the streaming version), or packed back into a file (for software versions).

The problem with parallelising this operation is that we need to know the lengths of all preceding code-words to know where a particular output code-word should appear in the output.

Without parallelism, this stage is prohibitively slow. In the worst case, this operation becomes bit-serial rather than just band or pixel serial, and this gives unacceptable performance. Fortunately, a solution exists. See Figure 2. Observe that we do not strictly require that all preceding packing operations have taken place before packing an output, only that we know the offset into the output stream of our word-to-be-packed.

Our encoder will tell us how many bits each output code-word contains, so if we now take a running sum of code-word lengths, this will give us the offset into the output stream for each output code-word.

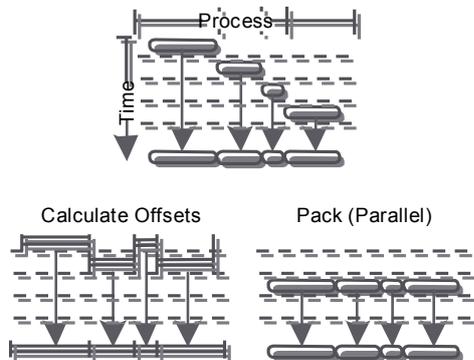


Figure 2 - Parallel packing of variable length code-words

Addition is commutative (in absence of saturation operations like those used in the predictor), so we can perform a running sum operation on the output word lengths using a tree structure. A fully parallelised add-tree operation will take  $\log(N)$  stages, where  $N$  is the number of items to be summed.

With the output offsets calculated, all of the output code-words can be shifted and moved into place with a logical-OR

operation, and these can be performed in parallel across all bands and spatial pixels.

It should be noted that, due to the compression itself, it is inevitable that multiple output code-words will reside in the same machine word of the output. Careful synchronisation will be needed to ensure that such writes occur atomically between processes. While we make numerous optimisations during implementation, we have not altered the algorithm in any way.

## III. IMPLEMENTATION OVERVIEW

Although the implementations described here do not rely on any specific architectural features found only on laptop computers, we shall still refer to these as the *Mobile CUDA* and *Mobile OpenMP* Implementations, to distinguish them from previous work. We shall refer to the implementations in [3] as the *Fixed Platform CUDA/OpenMP* versions.

The parallelization approach used for the Fixed Platform Implementations is well described in [3], but it is useful to summarize here – since some features are common to the new implementation.

### A. Fixed Platform CUDA/OpenMP Implementations

The parallelism approach taken in this implementation was to buffer data at any stage of the algorithm where a data dependency occurs. The buffers are taken to be as large as possible, and spaced to expose the maximum possible degree of algorithmic parallelism at each stage of computation.

The enormous benefit that this buffering brings is that there is never a point where one parallel execution thread is waiting on data from another – since each algorithm stage only begins when all of the data that it requires have been calculated.

This approach is essentially a ‘greedy optimization’; it takes the biggest parallel bites first. What remains at the end is a core surrounding the stage whose data dependencies cannot be removed, and this core proves to be the parallelism bottleneck.

Figure 3 shows how the algorithm is divided up horizontally into stages. Between each numbered stage is a data buffer, and stages are executed in sequence. The first column lists the order of execution, the second column describes the operation performed, the third column importantly describes the amount of available parallelism, and the final two columns show un-parallelizable system calls.

Clearly, stages 2, 3, 5, 7 have the highest degree of parallelism available – they are parallelizable over all 3 data dimensions. Stage 6 contains a tree-based summation, and has logarithmic performance; better than serial, somewhat worse than the most parallel parts. The amount of work that the stage contains is low, however, and so this stage is not the bottleneck.

The choke point is stage 4 – the predictor. Unfortunately, this is also the stage which contains the most computational work,

and so forms the performance bottleneck for the entire algorithm.

The buffers used end up being so large that care needs to be taken that they fit in GPGPU main memory, which is 1.5 GB on the test system. For this reason, the image is split into segments prior to processing. These segments are processed serially by the algorithm.

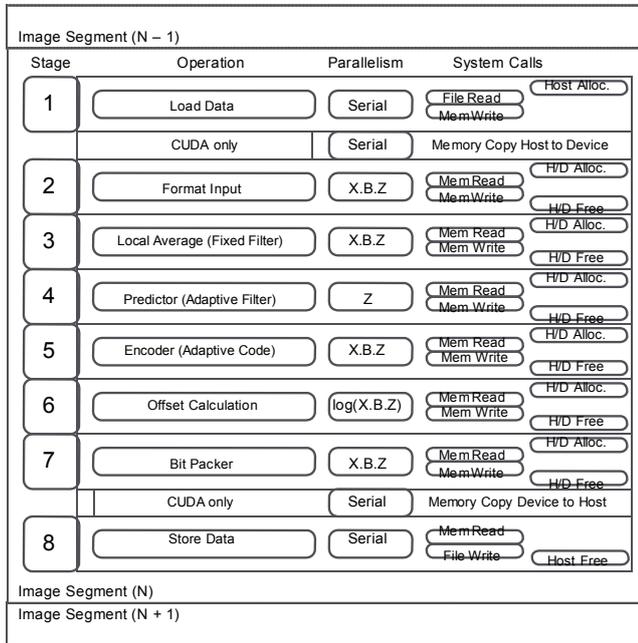


Figure 3 - Fixed Platform OpenMP / CUDA Implementation: Execution Timeline. Columns show order of algorithm stage execution, operation name, amount of available parallelism, and serial system calls

Although the Fixed Platform implementation is described here in terms of GPGPU, the OpenMP (multicore CPU) implementation is identical in architecture, operating sequentially on image ‘blocks’, and as a series of independent algorithm stages separated by large memory-resident buffers.

The key observations that allow the realization of the speedup in the Mobile implementation are:

1. Making the fastest stages of the algorithm faster comes at the expense of making the slowest parts even slower.
2. The model of streaming data through processing units from one memory resident buffer to another makes poor use of cache, and gives little opportunity for data-reuse.
3. The size of the buffers used mean that we cannot exploit additional sources of parallelism, like the image segmentation described above.

### B. Mobile CUDA Implementation

The key idea behind the new CUDA implementation is that restricting the parallelism of the highly parallel stages to

match that of the least parallel stage allows the stages to be merged. With all stages the same ‘width’, we no longer need giant buffers in memory to hold the output of each stage, and so data flow from one stage to the next carried in registers.

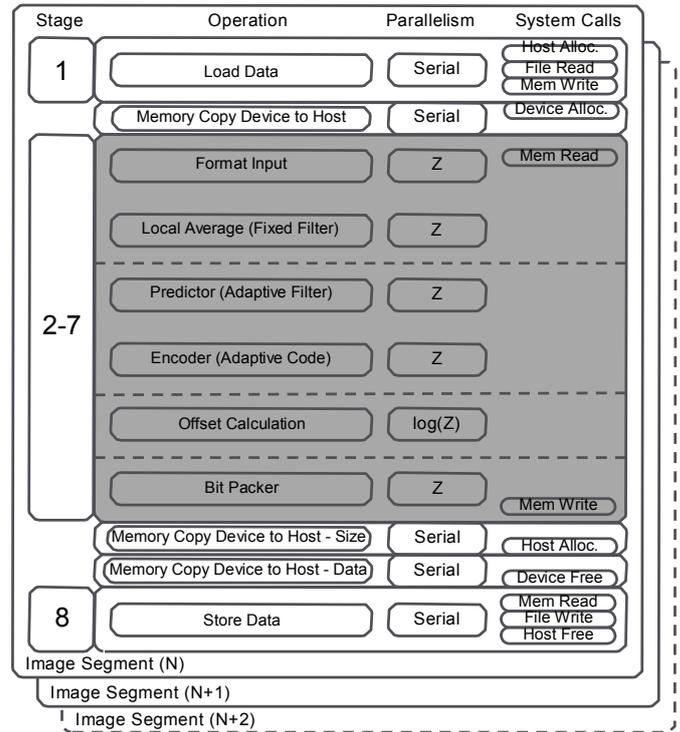


Figure 4 - CUDA Version 4 Architecture

The original purpose of the buffer was to break data-dependencies, and these then require careful synchronisation to avoid. We need to be able to share data between separate execution threads, and halt the execution of some threads while others ‘catch up’. The cost of requiring synchronisation turns out to be low, and we gain a side-benefit in the form of greater data-reuse. Suddenly the high-speed caches and shared memories on the GPGPU become usable. See TABLE I. for relative performance figures for caches / memories.

TABLE I. RELATIVE PERFORMANCE OF RAM TYPES AND BUSES

Device	Bandwidth	Latency
Disk – R/W	200 MB/s	-
Main Memory	3-17 GB/s	167
CPU L3 cache (6MB)	111 GB/s	30
CPU L2 cache (4x256kB)	245 GB/s	11
CPU L1 cache (4x32kB)	396 GB/s	5.5
PCIe (x8)	2.3 GB/s	-
GPU RAM (1.5 GB)	45 GB/s	> 1000
GPU L2 Cache (768 kB)	-	365
GPU Shared / L1 Cache (64 kB)	< 1.6 TB/s	88
GPU Register	< 8 TB/s	-

Instead of the algorithm being bound by the rate we can stream data into and out of memory, we can now push the limits of the computing capability of the GPGPU.

The other significant source of increased performance is the ability to leverage the parallelism created by image segmentation to boost the overall parallelism to a level more comfortable for the GPGPU.

In summary, the Mobile CUDA implementation exploits both spectral parallelism and spatial parallelism (in the form of image segmentation)

### C. Mobile OpenMP Implementation

While the Fixed Platform OpenMP (multicore CPU) implementation used exactly the same architecture as its CUDA counterpart, the Mobile OpenMP and CUDA versions look very different. The Mobile CUDA implementation makes use of both the band-parallelism inherent in the algorithm, and the imposed parallelism of image segmentation.

Surprisingly, the best performance is obtained from the OpenMP implementation when the band-parallelism is ignored entirely. The only source of parallelism the Mobile OpenMP implementation needs comes from the image segmentation. We effectively run multiple serial copies of the compression algorithm, each on its own processor core, and each independently compressing a different part of the image.

It should be noted that using the multiple CPU cores in parallel along the band axis is still significantly faster than a single core compressing data serially. However, the problem scales poorly with increasing numbers of cores – when using this type of band-parallelism.

There are a number of factors which together explain this result:

1. Multicore systems are not able to use the same high degree of parallelism as GPGPUs.
2. The cost of synchronisation between execution threads is much higher for multicore.
3. Sharing data between processors is more expensive, time-wise, than for GPGPU.
4. The highest level of caching available to data which is shared between cores is Level 3.

In the final stages of development it was realized that some of spectral-band parallelism could be reintroduced without causing inter-processor bottlenecks. Processors manufacturers keep adding new instruction sets to their products – and many of the additions are SIMD instructions to accelerate things like software video decoding. We can use a combination of the Intel SSE3 instructions, and a few operations from the newer SSE4 set. These instructions operate on 4x32-bit words, packed into special 128-bit SIMD registers. There are hardware instructions which can be used to explicitly control caching, and to stream data from memory through the SIMD registers in a DSP-like fashion. Intel produces a development package as part of its Parallel C++ Compiler suite called Intel

Performance Primitives (IPP). IPP is a library of accelerated functions for DSP, compression, and encryption which encapsulate the SSE instructions (emulating any which lack hardware support).

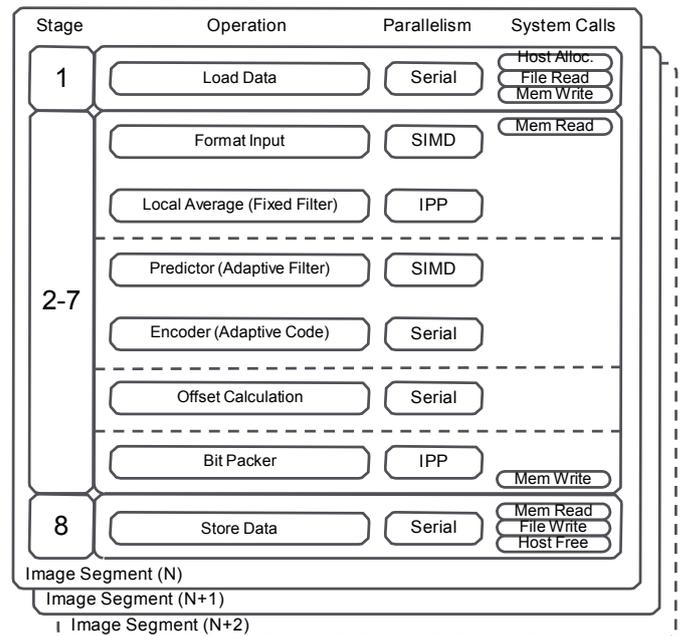


Figure 5 - OpenMP Version 4 Architecture

The computationally intensive kernel of the OpenMP multicore implementation was rewritten using a combination of IPP calls, and raw SSE instructions to add back a layer of spectral-band parallelism within a single core.

## IV. DECOMPRESSOR

An extra features added in the Mobile Platform Implementation is a multicore accelerated decompressor.

Decompression consists of two parts

1. Unpacking and decoding the variable length codewords.
2. Decompression of decoded data back to samples, using a matched predictor.

Stage 1 is the direct inverse of the bit-packer operation, and stage 2 is largely identical to the predictor in the compressor.

We would like to be able to use the same image segmentation technique to operate on multiple regions of image simultaneously. However, the variable length of compressed code-words causes a problem – we cannot know how where each compressed image segment begins in the compressed data. One solution is to add extra data at the compression stage, indicating the run-length of each compressed image segment. This enables the decompressor to ‘skip-ahead’, and identify the start position offsets of all image segments. Decompression can now proceed with the same level of parallelism as compression.

This technique is functionally equivalent to simply treating the original image as several smaller images and compressing

each individually. The compression of each sub-image is still entirely standards compliant.

It should be noted that the unpacking operation within an image segment cannot be parallelised. Therefore, were we to exploit band-parallelism in the predictor stage of the decompressor; we would still be left with a large serial portion of execution time. An OpenMP predictor for the decompressor would exhibit the same problems in exploiting band-parallelism as in the compressor. It is therefore not worth implementing a decompressor with any greater parallelism than image-segment spatial parallelism.

As highlighted above, the unpacker half of the decompressor cannot be parallelised at the band level, therefore would perform extremely poorly on GPGPU. Therefore a GPGPU accelerated decompressor is technically unfeasible.

## V. DEVELOPMENT AND TESTING ENVIRONMENT

In every test, the same input file and compression parameters were used. The output files for each implementation were checked and found to be bit-identical.

TABLE II. SPECIFICATION FOR HAWAII TEST IMAGE. SEE REF [11]

Height	512
Width	614
Bands	224
Bit Depth	12-bit unsigned
Byte Ordering	Big-Endian
Data Ordering	BIP
Uncompressed Size (kB)	137,536
Compressed Size (kB)	25,935
Compressed Size (%)	18.86%
Compressed Bit Depth (bpp)	<b>3.02</b>
Compression Ratio	<b>5.3 : 1</b>

To reduce the variance of test results, a tool called GameBooster3 [6][12] was used to shut down background processes (and to restart them after the test). This tool is freely available, and made a big improvement to the consistency of results – especially for the OpenMP implementations.

TABLE III. SPECIFICATION FOR MOBILE TEST PLATFORM

Manufacturer	Dell
Model	Alienware M18x
Processor	Intel Core i7-2760QM
Processor Clock	2.4GHz (4 Cores)
CPU Power (TDP)	45W
Chipset	Intel Sandy Bridge
System RAM	16GB DDR3 @ 666 MHz (PC3-10700)

Graphics Device	2 x Nvidia GeForce 560M GTX
GPU RAM (per device)	1.5GB GDDR5 @ 1.25 GHz
GPGPU Streaming Multiprocessors	6 @ 770 MHz
GPGPU Concurrent Threads	192
GPGPU Power (TDP)	75W (per device)
Hard Disk Devices	2 x 500 GB Seagate Momentus XT Hybrid SSD / Magnetic – Raid 0 (Striped)

## VI. IMPLEMENTATION RESULTS

The new implementations were developed to try and reach a throughput figure of 800Mb/s, corresponding to the real-time rate at which data are produced by a typical sensor.

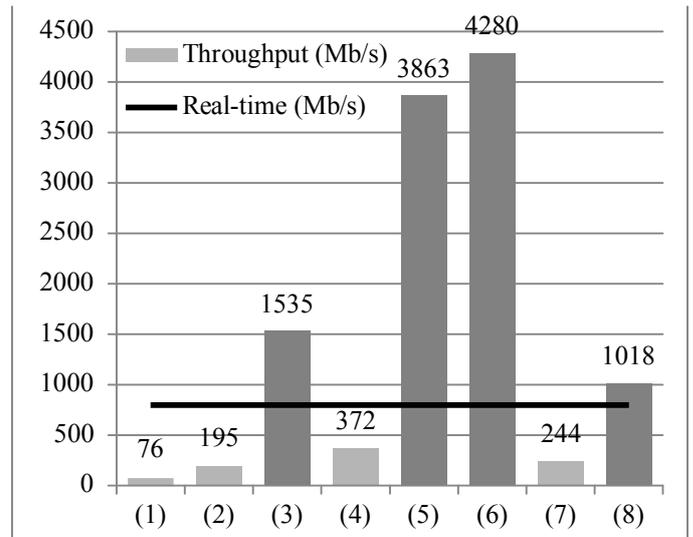


Figure 6 - Chart of throughput (in Mb/s) for different implementations. Dark Bars indicate new Mobile Platform Implementations, Light Grey indicate original Fixed Platform Implementations (1-3) OpenMP, (4-6) CUDA, (7-8) Decompressor. Horizontal line indicates real-time performance target.

The architectural and algorithm improvements developed allowed us to reach and comprehensively exceed this performance figure in both GPGPU accelerated, and un-accelerated multicore processor implementations.

TABLE IV. OPENMP IMPLEMENTATIONS PERFORMANCE

Version	Time (ms)	Throughput		Speedup
		(Mb/s)	(MSamp/s)	
1) Fixed Platform OpenMP - 1 Core	11542	75.64	5.82	-
2) Fixed Platform OpenMP - 4 Cores	4488	194.53	14.96	1.00
3) Mobile Platform OpenMP - 4 Cores	569	<b>1534.68</b>	118.05	7.89

The Mobile Platform OpenMP implementation shows a dramatic speedup over the original Fixed Platform implementation. Compression performance on a high-

performance workstation is expected to be slightly higher. The image segmentation added in the Mobile Platform implementation allows a laptop to compress data at nearly twice real-time throughput, without using any form of GPGPU acceleration.

The 4-core Fixed Platform Implementation exhibits only a 2.6x speedup compared with a single core implementation. This is most likely due to the Speedboost single-core overclocking.

TABLE V. CUDA IMPLEMENTATION PERFORMANCE

Version	Time (ms)	Throughput		Speedup
		(Mb/s)	(MSamp/s)	
4) Fixed Platform CUDA - 1 GPU	2346	372.14	28.63	1.00
5) Mobile Platform CUDA - 1 GPU	226	<b>3862.97</b>	297.15	10.38
6) Mobile Platform CUDA - 2 GPUs	204	<b>4279.56</b>	329.20	11.50

The increased parallelism gained by image segmentation, coupled with improved data-flow gives an enormous performance improvement on the Mobile Platform implementation.

In addition, the Mobile Platform implementation can make use of multiple GPGPU devices installed on the same system to increase performance further, but the performance gained from doing so is far less than double. The CUDA portion of the code is so fast, in this implementation, that the serial operations (file access / bus transfers) dominate the run-time, meaning we are close to the theoretical parallel performance limit for this algorithm.

TABLE VI. DECOMPRESSOR PERFORMANCE

Version	Time (ms)	Throughput		Speedup
		(Mb/s)	(MSamp/s)	
7) Decompressor OpenMP - Serial	3585	243.53	18.73	1.00
8) Decompressor OpenMP - Parallel	857	<b>1018.16</b>	78.32	4.18

A decompressor was also developed, targeted to multicore implementation which is able to exploit parallelism in the decoding stage, allowing data to be decompressed almost as fast as it is compressed, again beating the real-time performance target.

Decompression is slower than compression, due to an asymmetry in the bit-packing / unpacking operations. We show here how, by adding a small amount of extra data at the compression stage to the compressed file (or by transmitting several smaller output files), we can enable decompression to take place in parallel as well, greatly improving decompression performance. With this change, we see a 4-fold speedup by enabling image-segment parallelism in the decompressor, which is consistent with the speedup expected from a 4-core system.

## VII. CONCLUSIONS & FURTHER WORK

We have shown here that this hardware-optimised and apparently serial compression algorithm can be parallelised and optimised for software - exceeding the real-time performance barrier for the first time.

Further, we have demonstrated the suitability of this algorithm for implementation in software on commodity computing resources, in particular portable devices such as laptops.

As mentioned in the introduction, the FPGA implementation [2] does not make direct use of either the band or image-segmentation parallel described here. Both types of parallelism would be suitable, in some form, for hardware implementation. With the recent release of a radiation hard Virtex 5 FPGA device by Xilinx, it would be useful to update the hardware implementation to this technology, and build in parallel acceleration at the same time, allowing the same real-time performance to be achieved by space applications.

## ACKNOWLEDGMENT

Part of the research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The other part of the research described in this publication was carried out at the University of Edinburgh. Ben Hopson from University of Edinburgh would like to thank Wolfson Microelectronics Plc for funding his research.

## REFERENCES

- [1] W. Campbell, N. M. Short, "Remote Sensing Tutorial", 2004 [http://www.fas.org/irp/imint/docs/rst/Sect13/Sect13\\_9.html](http://www.fas.org/irp/imint/docs/rst/Sect13/Sect13_9.html)
- [2] N. Aranki, D. Keymeulen, A. Bakhshi and, M. Klimesh, "Hardware Implementation of Lossless Adaptive and Scalable Hyperspectral Data Compression for Space", In NASA/ESA Conference on Adaptive Hardware and Systems, IEEE, July 2009.
- [3] D. Keymeulen, N. Aranki, B. Hopson, A. Kiely, M. Klimesh, and K. Benkrid "GPU Lossless Hyperspectral Data Compression System for Space Applications", In IEEE Aerospace Conference 2012, IEEE, March 2012 (in press)
- [4] CCSDS, Lossless Multispectral & Hyperspectral Image Compression (Draft Recommended Standard) vol. 123.0-R-1: CCSDS, 1997-2007. (<http://public.ccsds.org/sites/ewerids/Lists/CCSDS%201230R1/Attachments/123x0r1.pdf>)
- [5] S. W. Golomb, "Run-length encodings," IEEE Trans. Information Theory, vol. IT-12, pp. 399-401, July 1966
- [6] Memon, N. "Adaptive coding of DCT coefficients by Golomb-Rice codes" in Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on
- [7] Aaron Kiely, "Selecting the Golomb Parameter in Rice Coding", JPL IPN Progress Report 42-159. November, 2004 [http://ipnpr.jpl.nasa.gov/progress\\_report/42-159/159E.pdf](http://ipnpr.jpl.nasa.gov/progress_report/42-159/159E.pdf)
- [8] R. Farber. CUDA Application Design and Development. Morgan Kaufmann, 2011
- [9] J. Sanders. CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison Wesley, 2010
- [10] D.B. Kirk. Programming Massively Parallel Processors : A Hands-on Approach (Applications of GPU Computing Series). Morgan Kaufmann, 2010.
- [11] Aaron Kiely, AVIRIS Hawaii Scene 1, 2001, Flight f011020t01p03r05 <http://compression.jpl.nasa.gov/hyperspectral/>
- [12] IOBit Game Booster 3: <http://www.iobit.com/gamebooster.html>

