

GPU Lossless Hyperspectral Data Compression System for Space Applications

Didier Keymeulen¹, Nazeeh Aranki¹, Ben Hopson², Aaron Kiely¹, Matthew Klimesh¹, Khaled Benkrid²

¹Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Dr., Pasadena, CA 91109, USA

²The University of Edinburgh, Institute of Integrated Systems, King's Buildings, Mayfield Road, Edinburgh, EH9 3JL, UK
818-354-4280

didier.keymeulen@jpl.nasa.gov

Abstract—On-board lossless hyperspectral data compression reduces data volume in order to meet NASA and DoD limited downlink capabilities. At JPL, a novel, adaptive and predictive technique for lossless compression of hyperspectral data, named the Fast Lossless (FL) algorithm, was recently developed. This technique uses an adaptive filtering method and achieves state-of-the-art performance in both compression effectiveness and low complexity. Because of its outstanding performance and suitability for real-time onboard hardware implementation, the FL compressor is being formalized as the emerging CCSDS Standard for Lossless Multispectral & Hyperspectral image compression. The FL compressor is well-suited for parallel hardware implementation. A GPU hardware implementation was developed for FL targeting the current state-of-the-art GPUs from NVIDIA®. The GPU implementation on a NVIDIA® GeForce® GTX 580 achieves a throughput performance of 583.08 Mb/s (44.85 MSamples/sec) and an acceleration of at least 6 times a software implementation running on a 3.47 GHz single core Intel® Xeon™ processor. This paper describes the design and implementation of the FL algorithm on the GPU. The massively parallel implementation will provide in the future a fast and practical real-time solution for airborne and space applications.

images typically consist of hundreds of spectral bands; the voluminous amount of data comprising hyperspectral images makes them appealing candidates for data compression. An example of a hyperspectral data cube is shown in Figure 1. It was taken by the Airborne Visible and Infrared Imaging Spectrometer (AVIRIS), which uses diffraction gratings for band separation with two sets of CCD arrays, one with silicon chips to sense in the visible range and the other with Indium-Antimony (InSb) chips for wavelengths in the Near-IR to Short-Wave-IR range. AVIRIS has 224 detectors (channels) in the spectral dimension, extending over a range of 0.38 to 2.50 μm . This arrangement leads to a spectral resolution for each chip of 0.01 μm . The spatial resolution derived from this depends on the platform height. A typical mission, mounting AVIRIS on a NASA aircraft (ER-2), produces a spatial resolution of about 20 meters, but this can be improved to five meters by flying at lower altitudes, which, of course, narrows the width of the ground coverage [1].

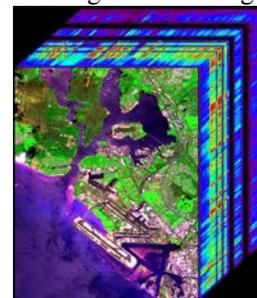


Figure 1: An example of a hyperspectral data cube for Pearl Harbor, Hawaii taken by the AVIRIS instrument

TABLE OF CONTENTS

1. INTRODUCTION	1
2. ADAPTIVE FILTERING	2
3. GPU IMPLEMENTATION	4
4. PERFORMANCE	6
5. FUTURE WORK	7
6. SUMMARY	7
ACKNOWLEDGEMENTS	7
REFERENCES	7
BIOGRAPHY	8

1. INTRODUCTION

Hyperspectral images are three-dimensional data sets, where two of the dimensions are spatial and the third is spectral. A hyperspectral image can be regarded as a stack of individual images of the same spatial scene, with each such image representing the scene viewed in a narrow portion of the electromagnetic spectrum. These individual images are referred to as spectral bands. Hyperspectral

Current NASA hyperspectral instruments either avoid compression or make use of only limited lossless image compression techniques during transmission. For example, the current state-of-the-practice is to use the Universal Source Encoder for Space (USES) chip [2]. USES implements the lossless compression standard [3] proposed by the Consultative Committee for Space Data Systems (CCSDS), which is based on the Rice algorithm. USES includes a multispectral mode to extend its operation to 3D data sets. The USES chip achieves limited compression effectiveness compared to other existing techniques, but has the advantage of being currently available in a radiation resistant form. The main reasons for utilization of such devices by NASA are: the limited downlink bandwidth, the need to reduce the risk of corrupting the data-stream needed for accurate science processing, and the lack of a

viable on-board platform to perform significant image processing and compression. Future instruments with more sensors and a much larger numbers of spectral bands will collect enormous volumes of data that will far outstrip the current ability to transmit it back to Earth (data rates for some instruments can go to several hundreds of Gbits/s). This gives rise to the need for efficient on-board hyperspectral data compression.

Exploiting dependencies in all three dimensions of hyperspectral data sets promises substantially more effective compression than two-dimensional approaches such as applying conventional image compression to each spectral band independently. With that in mind, the JPL Fast Lossless (FL) hyperspectral compressor was developed [4]. FL is a predictive technique that uses an adaptive filtering method and achieves state-of-the-art performance in both compression effectiveness and low complexity. Because of its outstanding performance and suitability for real-time onboard hardware implementation, the FL compressor is being formalized as the emerging CCSDS 123.0 standard for lossless multispectral and hyperspectral image compression [19], [20].

Beside its low complexity, the other main advantage of the FL algorithm is that it is well-suited for parallel hardware implementation. However, traditional general purpose processor (GPP) based software implementations of this algorithm have limited throughput performance and are power hungry. Dedicated hardware solutions are hence highly desirable, taking the load off the main processor while providing a power-efficient solution at the same time. VLSI ASIC implementations are power- and area-efficient, but they lack flexibility for post-launch modifications and repair, they are not scalable and cannot be configured to efficiently match specific mission needs and requirements. Field Programmable Gate Arrays (FPGAs) are programmable and offer a relatively low cost and flexible solution compared to traditional Application-Specific Integrated Circuit (ASICs). The development time of an FPGA solution however is much longer than that of a GPP-based solution as development of an FPGA solution is essentially a circuit design exercise. In recent years, Graphics Processing Units (GPUs) have been proposed and successfully harnessed as a high performance, low cost, relatively easy to program, and flexible computing platform for many applications beyond traditional graphics processing, giving rise to a new discipline called General Purpose GPU computing or GPGPU. A GPU essentially consists of a large number (hundreds) of parallel processors with a memory hierarchy that allows for the concurrent processing of thousands of threads. GPUs are generally programmed in a Single Program Multiple Threads (SPMT) fashion in which GPU processors (also called cores) execute the same program on different parts of the data using different threads. With unique thread IDs, the program could also make different threads execute different instructions. The key to the high performance of GPU solutions resides in the efficient mapping of applications

onto the underlying GPU architecture (multiprocessors and memory hierarchy), a task which while simpler than FPGA solution development is more difficult than traditional single-threaded GPP programming.

The most widely used architecture for GPGPU is CUDA (Compute Unified Device Architecture) from NVIDIA Corp, which is traditionally programmed in the C language with extensions that allow for the SPMT programming model. Typically, a host program running on a GPP first copies data from host memory to GPU memory; then the GPP initiates the processing on GPU; following which the GPU executes its program in parallel using its processor cores; and finally results are copied back from GPU memory to host memory.

Raw data from pushbroom-type hyperspectral imagers tends to exhibit streaking-artifacts parallel to the along-track direction. Options in the CCSDS standardization of the FL compressor allow the user to tailor compression to handle either data from pushbroom-type sensors or data that does not include such artifacts (e.g., calibrated imagery or data from whiskbroom-type sensors such as AVIRIS).

In previous work we have described an FL implementation for whiskbroom-type sensors [14] and one for pushbroom-type sensors [18]. In this paper, we describe the algorithm for pushbroom-type sensors and its GPU implementation that result in improved compression speed-up when the algorithm is applied to raw (uncalibrated) data from a pushbroom-type multispectral imager [15].

The remainder of this paper is organized as follows. Section 2 describes background on the hyperspectral compression algorithm used in this paper. Then, section 3 describes our GPU-based implementation of the hyperspectral compression algorithm and its trade-offs. After that, section 4 describes the results of our initial experiments. Section 5 follows with plans for future work. The paper concludes with a summary.

2. ADAPTIVE FILTERING

Pushbroom-type Instruments

Pushbroom-type multispectral imagers use a detector array to acquire data in spatial-spectral slices. Thus each detector element corresponds to a specific spectral band and cross-track position. Because the characteristics of detector elements generally vary somewhat from element to element, cross-track adjacent samples in a given spectral band will not be as similar as they would be in an instrument that uses the same detector element for all samples in a given spectral band (e.g., in a whisk-broom-type instrument). On the other hand, along-track adjacent samples will tend to be very similar. As a side effect of the variation within spectral bands, the correlation between samples at the same spatial location in different spectral bands varies with cross-track position. As such, purely spectral prediction often does not work well with this type of data. Pushbroom-type instruments are generally the multispectral imagers of choice for space applications (as

opposed to whisk-broom-type instruments). In the following sections we describe both the Fast Lossless compressor as it was originally conceived, as well as the modification intended for pushbroom instruments.

Algorithm Background

The Fast Lossless compressor encodes data samples one-at-a-time, typically in raster scan order within a given spectral band. It uses a form of predictive compression, i.e. sample values are estimated by linear prediction, and the differences between the estimates and the actual sample values are encoded into the compressed bitstream. Only previously encoded samples are used to predict a given sample so that the prediction operation can be duplicated by the decoder. Estimation of sample values by linear prediction is a natural strategy for lossless compression of hyperspectral images. This is a form of predictive compression, or, more specifically, a form of differential pulse code modulation (DPCM).

The Fast Lossless compressor uses the sign algorithm [5], which is a variation of the Least Mean Square (LMS) algorithm [6], a well-known low-complexity adaptive filtering algorithm. The sign algorithm and the LMS algorithm are members of a family of low complexity adaptive linear filtering techniques. The literature includes a fair amount of other work on lossless predictive compression of hyperspectral images. For example, the methods used by Rizzo et al. [7] have low complexity and yield compression effectiveness similar to that of FL. Good compression effectiveness results are also reported in the literature by Aiazzi et al. [8], but those results are obtained with methods of moderately high complexity.

Fast Lossless Algorithm Description

The essence of the Fast Lossless compressor is adaptive linear predictive compression using the sign algorithm for filter adaptation, with local mean estimation and subtraction. We start with a brief description of the LMS algorithm and the sign algorithm. For both of these algorithms a desired signal d_t is to be estimated from an input (column) vector $u_{t,k}$, where t is an index which increases sequentially and represents the time index. The desired signal d_t is the sample value at spatial location (x, y) in spectral band z . The estimate \hat{d}_t is a linear function of $u_{t,k}$; specifically, $\hat{d}_t = w_{t,k}^T u_{t,k}$, where $w_{t,k}$ is the filter weight vector at index t . The components of $u_{t,k}$ represent the sample values at spatial location (x, y) in spectral band $z-k$ with $z=1,2$ and 3 , as well as the sample values at neighborhood location $(y-1,x-1)$, $(y-1,x)$, $(y,x-1)$ in spectral band z .

After an estimate \hat{d}_t is computed, the error between

the estimate \hat{d}_t and the desired signal d_t is computed, specifically, $e_t = \hat{d}_t - d_t$.

This error value is used to update the filter weights. For the LMS algorithm,

$$w_{t+1,k} = w_{t,k} - \mu u_{t,k} e_t$$

For the sign algorithm,

$$w_{t+1,k} = w_{t,k} - \mu u_{t,k} \text{sgn}(e_t)$$

In each case μ is a positive, scalar parameter (the step size parameter) that controls the trade-off between convergence speed and average steady-state error. A small μ results in better steady state performance but slower convergence. In some variants of these algorithms the value of μ changes over time. The sign algorithm has the property that under certain general assumptions, the weight vectors it produces become clustered around the optimum weight vector in terms of minimizing the mean absolute estimation error. For a sufficiently small adaptation step size parameter, the asymptotic mean absolute estimation error can be made to be as close as desired to the minimum possible [5].

To overcome problems of poor combinations of convergence speed and steady-state performance, a local mean subtraction method was used, motivated by [9]. In the local mean subtraction method, for each sample we compute a preliminary estimate using a fixed, causal, linear predictor involving only samples from the same band (purple cells in Figure 2). The preliminary estimate of sample $s(x, y, z)$ is denoted by $\tilde{s}(x, y, z)$ which is the local mean of sample values at $(y-1,x-1,z)$, $(y-1,x,z)$, $(y,x-1,z)$ and $(y-1,x+1,z)$. For our implementation we use a six-sample prediction neighborhood with three samples from the same band as the sample to be predicted, and one sample each from the three preceding bands (blue cells in Figure 2).

All samples are corrected using the local mean subtraction method so that:

$$u_{t,k} = \begin{bmatrix} s(x-1, y-1, z) - \tilde{s}(x, y, z) \\ s(x, y-1, z) - \tilde{s}(x, y, z) \\ s(x-1, y, z) - \tilde{s}(x, y, z) \\ s(x, y, z-1) - \tilde{s}(x, y, z-1) \\ s(x, y, z-2) - \tilde{s}(x, y, z-2) \\ s(x, y, z-3) - \tilde{s}(x, y, z-3) \end{bmatrix} = \begin{bmatrix} \text{Diff 3} \\ \text{Diff 2} \\ \text{Diff 1} \\ \text{Diff 4} \\ \text{Diff 5} \\ \text{Diff 6} \end{bmatrix}$$

is the corresponding input vector. The general rule is to adjust each sample in the prediction neighborhood by the preliminary estimate in the same band as the sample but at the spatial location of the sample being predicted.

Handling Pushbroom Data

As explained above, for each sample a local mean is computed as the average sample value of four adjacent samples in a causal neighborhood within the spectral band [15]. However, for data from pushbroom-type multispectral imagers, letting the local mean be equal to the previous sample in the same cross-track position (and in the same

spectral band) gives significantly better results.

Specifically, to accommodate raw data from pushbroom imagers, for all rows (along-track positions) except the first row in a segment, we let the local mean equal the previous sample in the same cross-track position, specifically position $(y-1, x, z)$. For the first row in a segment, no such sample is available, so we let the local mean equal the causal cross-track adjacent sample, specifically position $(y, x-1, z)$. In addition, the prediction neighborhood of a sample is changed. In the original algorithm described above, this neighborhood contained three samples from the same band as the given sample and one sample from each of the three preceding spectral bands; under the modified algorithm, only the three samples from three preceding spectral bands are used.

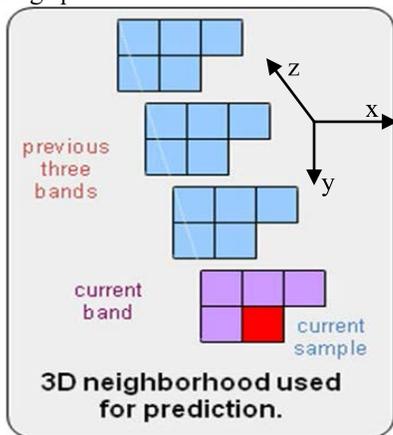


Figure 2: 3D prediction neighborhood

Golomb Encoder and Decompressor

The difference is encoded in the compressed bit stream using Golomb codes [10] with parameters that are powers of 2. The decompressor decodes this difference from the bitstream, and from previously decoded samples, and therefore can reconstruct the value $s(y, x, z)$. Further details of the algorithm can be found in [4] and [15].

Compression Performance

The Fast Lossless algorithm provides outstanding compression effectiveness. Tests with uncalibrated AVIRIS data sets demonstrate compression results of about 40% lower bit rate than state-of-the-art 2D approaches (approximately 4:1 compression ratio) as shown in Figure 3. The algorithm also performs well compared to more complicated 3-D algorithms such as ICER-3D [11][12][13].

3. GPU IMPLEMENTATION

To explain how best to parallelize this algorithm, we need to look at the data-dependency graph. Figure 4 shows an outline for the algorithm. The following sections will take each important block in turn, and discuss the implied GPU implementation details.

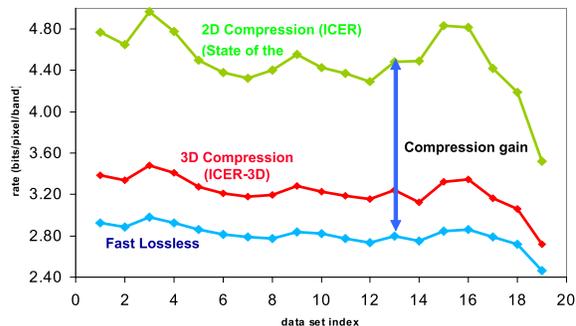


Figure 3: Compression performance average over 19 uncalibrated AVIRIS hyperspectral test data sets. ICER and ICER-3D are wavelet-based 2D image and 3D hyperspectral compressors developed at JPL.

Data Formatting

Data ordering is extremely important to the performance of the GPU implementation. We want to arrange the data so that as many threads as possible access memory locations which are close together so that multiple data samples can be obtained from the same cache read. Since we are parallelizing along the band axis, it makes sense that adjacent bands of a single image sample are adjacent in memory. In other words, whatever the data format in file, we want to store it in memory in Band-Interleaved-Pixel (BIP) format.

To save space in the instrument, data is output as packed 16-bit words. Before any processing can occur, we have to unpack to the GPU native 32-bit integer format. This is an excellent point to permute bytes to handle endian-ness. Helpfully, CUDA provides instruction support in the form of the `__byte_perm` instruction, which is a combined permute / unpack instruction, for working with byte-packed data.

Local Average

The CCSDS standard [19] gives a choice of methods: *column-oriented local sums* (for use with *pushbroom* sensors), or *neighbor oriented local sums* (for *whisk-broom* sensors). Both can be seen as small kernel image convolution. In the column-oriented case, each output depends only on at most 2 inputs as described in the previous section. In the neighbor-oriented version, each average uses up to 5 input samples.

The memory access pattern for both is extremely simple, so there is no contention between threads. It would seem that the edge cases would need branching to handle, but this is in fact not the case. As long as the thread block size is a multiple of the number of bands, all block threads will be accessing different bands from the same image sample position at the same time. Therefore, the entire block will take the branch, in the case of an edge condition. This means that we don't have any thread divergence, and we don't need a halo region for the image.

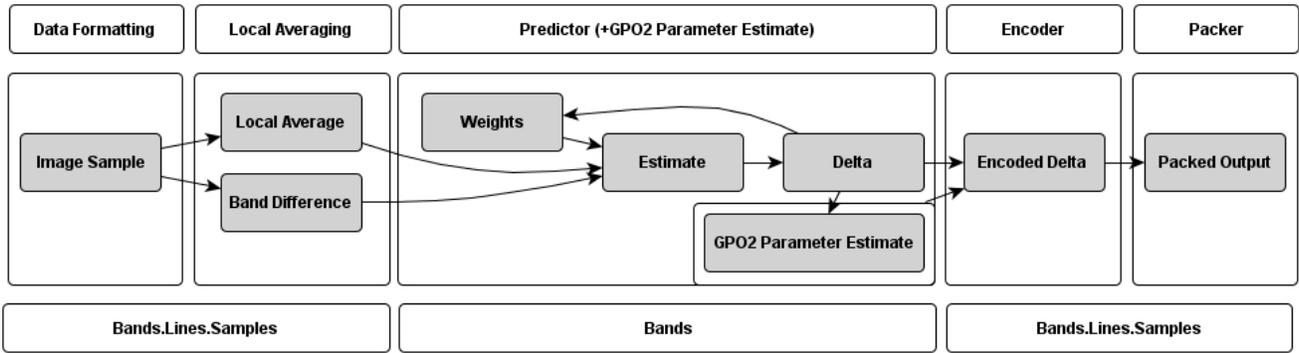


Figure 4: Algorithm Stages / Data Dependency / Available Parallelism: The top line of the figure gives the block names, the middle layer shows the data dependency, and the bottom layer lists the available parallelism.

Since there is no interdependency between separate average values, we can parallelize this section across the full volume of the image. That is, the averaged value of any two pixels can be calculated in parallel. Restricting the threads within a block to all lie within the same pixel (across 224 bands), we maximize memory access locality.

Predictor

The predictor, due to the feedback caused by the weight update, has to process pixels serially. Parallelism along the band axis is possible however, but this is still the bottleneck of the algorithm. There is a subtle trade-off to make here, between reducing the amount of work done in the serial loop so that individual loop iterations take as little time as possible, and packing enough calculation into each loop so that the GPU has something to do while fetching from RAM. Normally, we would have enough threads that threads blocking on memory access would immediately be swapped out for ones whose data was available (trading off computation against memory bandwidth).

We are limited in how many threads we can have (limited to the number of spectral bands), so we need to make sure that threads do as much work as possible for each memory read. For this reason, it actually turns out optimal to include the Golomb Power-of-2 (GPO2) code parameter estimator into the same loop as the predictor – even though the data that this calculation needs could be pre-calculated by the predictor, stored, and processed in a separate (much more parallel) kernel invocation. (Adding the estimator to the predictor barely increased the run time of the predictor stage).

GPO2 Parameter Calculation.

To dynamically determine the GPO2 parameter k we need to compute the base-2 log of a running average of the values to be encoded ($delta$), and this will give us a bound. That is, we need the smallest k such that 2^k is greater than the running average $delta$ value. Fortunately, CUDA comes to aid again, with the `__nlz` instruction, which counts the number of leading zeros of an integer. Equipped with this, it is trivial to compute the log, and extremely fast.

Encoder

The encoding stage is extremely simple. It takes in a $delta$ value to be encoded, and the code parameter k . It is then just a matter of a few bit operations to derive the output codeword. Since output codewords have variable width, we need to record internally the output codeword length for use in the bit-packer, which is discussed in the next section.

There is no dependency between the encoding of output samples, even across bands, once we have each $delta$ and its code parameter. It makes sense, therefore, to pre-compute these (remember, the code parameter calculation was rolled into the predictor) and run the encoder in parallel across bands and pixels.

Packer

The job of the bit packer is to take the variable length encoded $delta$ values and write these out serially into a bit-stream. While it looks like this is a classically unparallelizable process it can, in fact, be parallelized by first computing the indices in the output (packed) stream of each encoded delta value. This is simple: we take the lengths (in bits) of each encoded delta value and compute a running sum. While not fully parallel, a running sum (or scan-reduce) can be implemented as a tree structured process. The *THRUST* library (included in the latest CUDA GPU toolkit) has optimized routines for doing just this and these are extremely fast [16].

With the output stream offset computed for each encoded output delta, a single GPU thread has only to read one encoded output value, split it (where the value strides a 32-bit word boundary) across two output words, and *bit-or* the encoded value into the final output stream. The *or* operation must be performed atomically to prevent data-corruption. This causes a performance issue on pre-Fermi architectures, but the Fermi's caching removes this issue since we only need read/write atomically to shared RAM (the Fermi uses thread-shared memory for its cache) instead of GPU global RAM.

Since each output value has at least 1 bit, were we to modify the thread indexing so that encoded values are read

on a stride of 32, we would never have more than one thread accessing the same output word. That is to say: we pack encoded (but not packed) blocks 0,32, 64, etc on the first pass through the data – which guarantees that no two threads write to the same 32-bit output word in the same data-pass and so on for blocks 1, 33, etc on the next pass. This removes the need for an atomic operation, but destroys the cache coherence. On pre-Fermi architectures (compute capability 1.3 and lower) this trade-off is worth making, but on the Fermi (compute capability 2.0 onwards) it is not worth it.

By breaking out the serial portion (the cumulative sum operation) and running it first, we can run the bit packer in full pixel & band parallelism.

Output Formatting

Finally, the output words are bit-reversed (hardware support to the rescue again with the fast `brev` instruction), and byte shuffled to correct big-endian / little-endian compatibility.

This has to be performed after the bit packer has completed (but the bit-reversal can occur during bit packing with a suitable modification to the shift and masking operations), but this stage can again be performed with the full parallelism available in the image data.

4. PERFORMANCE

The two main reasons for moving to GPU are speed, and to reduce the CPU loading. We have evaluated the performance on a mobile and desktop platform.

The mobile test system used was:

- a) Intel® Core™ i7 Processor 2760QM (base clock speed 2.40Ghz)
- b) NVIDIA® GeForce® GTX 560M (1.5GB RAM)

The desktop test system used was:

- c) Intel® Xeon™ Processor X5690 (base clock speed 3.46Ghz)
- d) NVIDIA® GeForce® GTX 580 (1.5GB RAM)
- e) NVIDIA® Tesla® C2070 (6GB RAM)

The test data was an uncalibrated AVIRIS hyperspectral image from Hawaii with dimensions 614 x 512, with 224 spectral bands, and with 12-bits per samples [17]. For this specific image, the output was compressed to 25% of the original packed size (3.02 bits per pixel, a factor 3.98 compression).

With the implementation described above, the CPU is only used for host memory allocation and file access (at both the input and output). Peak CPU utilization is typically less than 3% for the mobile implementation due to fast memory access thanks to the Solid State Drive used in the mobile platform, so this is a substantial benefit (Figure 5). Note that by using the CUDA support for non-paged host memory, very little CPU work is involved in the data-transport to and from the GPU.

Shown in Table 1 is a breakdown of time spent on each

stage of the algorithm on the mobile platform. Only the *File Load* and *File Store* stages use the CPU at all, demonstrating the low CPU utilization. Very clearly, the bulk of the time is taken in the predictor stage, as this is the stage with the least available parallelism. The time is slightly better than the FPGA implementation on a Virtex-4 presented in [18].

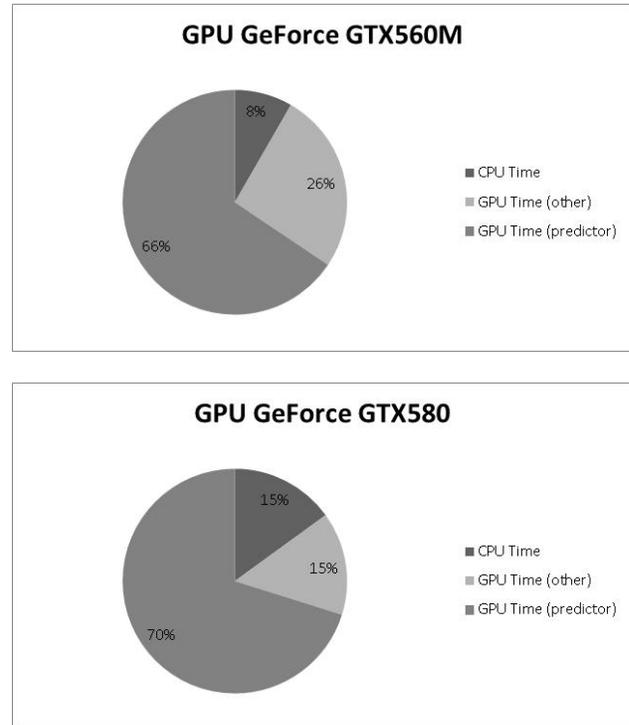


Figure 5 Timing breakdown between CPU and GPU of the GPU implementation of the Fast Lossless Compression on both the NVIDIA mobile board (GTX560M) and desktop board (GTX580).

Table 1: Timing breakdown by algorithm stage for GPU.

Algorithm Stage	GTX560M		GTX580	
	Total Time (ms)	Time per sample (ns)	Total Time (ms)	Time per sample (ns)
File Load (CPU)	48.24	0.69	86.88	1.23
Input Formatting	53.33	0.76	4.74	0.07
Local Average	70.79	1.01	30.21	0.43
Predictor & Entropy Estimator	1253.17	17.80	1116.69	15.86
Encoder	59.51	0.85	22.55	0.32
Output Index Sum	114.12	1.62	32.99	0.47
Packer	96.21	1.37	80.10	1.14
Output Formatting	103.97	1.48	65.27	0.93
File Store (CPU)	111.01	1.58	151.93	2.16
Total time	1910.35	27.13	1591.36	22.60

For comparison, the same algorithm recoded to run

without CUDA acceleration running on 1 and 4 CPU cores (Multicore version coded using OpenMP) both on the mobile and desktop platform is shown in Figure 6 and in Table 2.

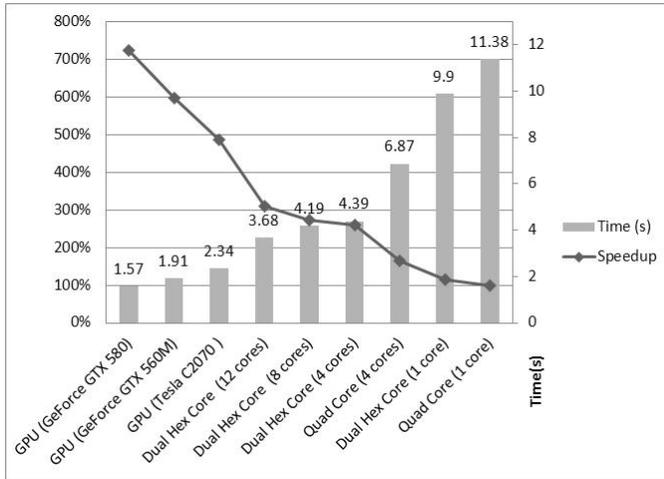


Figure 6: Speedup comparison of the Fast Lossless Compression Algorithm on both the mobile platform with GTX560M GPU and the dual hex core CPU (2.4GHz) and the desktop platform with the GTX580 and the Tesla C2070 GPUs and the quad code CPU (3.47GHz).

This result demonstrates that the GPU implementation could accommodate real-time compression for hyperspectral instrument which has normally throughput of 800 Mbits/sec e.g. through multiple GPUs or further code optimization.

Table 2: Speedup comparison of GPU against CPU implementations

	Speedup	Time (s)	Speed (Mbit/s)	Speed (MSamp/s)
GPU GeForce GTX 580	725%	1.57	583.08	44.85
GPU GeForce GTX 560M	596%	1.91	479.29	36.87
GPU Tesla C2070	486%	2.34	391.21	30.09
Dual Hex Core (12 cores)	309%	3.68	248.76	19.14
Dual Hex Core (8 cores)	272%	4.19	218.48	16.81
Dual Hex Core (4 cores)	259%	4.39	208.53	16.04
Quad Core (4 cores)	196%	6.87	133.25	10.25
Dual Hex Core (1 core)	115%	9.9	92.47	7.11
Quad Core (1 core)	100%	11.38	80.44	6.19

5. FUTURE WORK

Commonly one might divide an image into smaller pieces (*segments*, e.g., consisting of 32 lines of image data) and apply compression to separate segments independently (i.e., reinitializing the predictor and GPO2 parameter estimation for each segment). For example, this might be done to limit the effects of data losses if the compressed data is transmitted over a noisy channel. This type of segmentation gives another axis for potential

parallelization. It would be possible to use this to improve the GPU utilization of the bottleneck stage by allowing threads from separate blocks to be scheduled at the same time. Many high-end dedicated GPGPU platforms contain several GPU cards – the problem could easily be split task-parallel across separate GPUs, again using the blocks as the axis for parallelization. The multicore GPP implementation is under-tuned compared with the GPU version and needs to be improved.

It is difficult to estimate the performance gain that could be gained by some of these optimizations but splitting the problem task-parallel over two GPUs would give almost a factor 2 (the file access portions would not parallelize). It is harder to estimate the performance impact of, say, splitting the image into 16 blocks and operating on all the blocks in parallel on a single GPU. A factor 16 is unlikely, but a factor 2-4 is plausible (the multi-core GPP implementation showed roughly square root scaling with increases in available parallelism, for comparison).

6. SUMMARY

We presented a GPU implementation of the JPL-developed Fast Lossless multispectral and hyperspectral data compression algorithm, currently being formalized as an emerging CCSDS standard [19]. The implementation targets a desktop and mobile GPU hardware from NVIDIA®. For the desktop implementation, the NVIDIA® GeForce® GTX 580 provides an acceleration of at least 6 times the software implementation on a single core Intel® Xeon™ Processor (clock speed 3.47GHz) while for the mobile implementation, the NVIDIA® GeForce® GTX 560M provides an acceleration of at least 7 times the software implementation on a single core Intel® Core™ i7 Processor (clock speed 2.40GHz). These results make the use of this compressor practical for satellites and planet orbiting missions with hyperspectral instruments. Future development will provide multiple implementations on GPU and multicore GPPs and options to deploy various versions of the algorithm to accommodate data from different instrument types.

ACKNOWLEDGEMENTS

The work described in this publication was carried out at the Jet Propulsion Laboratory, California Institute of Technology and the University of Edinburgh.

REFERENCES

- [1] W. Campbell, N. M. Short, “Remote Sensing Tutorial”, 2004 http://www.fas.org/irp/imint/docs/rst/Sect13/Sect13_9.html
- [2] J. Venbrux, J. Gambles, D. Wiseman, G. Zweigle, W. H. Miller, and P.-S. Yeh, “A VLSI Chip Set Development for Lossless Data Compression,” *Ninth AIAA Computing in Aerospace Conference*, San Diego, California, October 19–21, 1993.
- [3] *Lossless Data Compression*, Recommendation for Space Data System Standards, CCSDS 121.0-B-1. Blue Book. Issue 1.

- Washington, D.C., CCSDS, May 1997. (<http://public.ccsds.org>)
- [4] M. Klimesh, "Low-Complexity Lossless Compression of Hyperspectral Imagery via Adaptive Filtering," *The Interplanetary Network Progress Report*, vol. 42-163, Jet Propulsion Laboratory, Pasadena, California, pp.1–10, November 15, 2005.
- [5] A. Gersho. "Adaptive filtering with binary reinforcement". *IEEE Transactions on Information Theory*, IT-30(2):191–199, March 1984.
- [6] B. Widrow, J. R. Glover, J. M. McCool, J. Kaunitz, C. S. Williams, R. C. Goodlin, J. R. Zeidler, R. H. Hearn, and E. Dong. "Adaptive Noise Cancelling: Principles and Applications". *The Proceedings of the IEEE*, 63(12):1692–1716, December 1975.
- [7] F. Rizzo, B. Carpentieri, G. Motta, and J. A. Storer. "Low-complexity lossless compression of hyperspectral imagery via linear prediction". *IEEE Signal Processing Letters*, 12(2):138–141, February 2005.
- [8] B. Aiazzi, L. Alparone, and S. Baronti. "Near-lossless compression of 3-D optical data". *IEEE Transactions on Geoscience and Remote Sensing*, 39(11):2547–2557, November 2001.
- [9] J. N. Lin, X. Nie, and R. Unbehauen, "Two-Dimensional LMS Adaptive Filter Incorporating a Local-Mean Estimator for Image Processing," *IEEE Transactions on Circuits and Systems—II: Analog and Digital Signal Processing*, vol. 40, no. 7, pp. 417–428, July 1993
- [10] R.G. Gallager and D.C. Van Voorhis. "Optimal source codes for geometrically distributed integer alphabets". *IEEE Transactions on Information Theory*, IT-21 (2): 228-230, March 1975.
- [11] A. Kiely, "Simpler Adaptive Selection of Golomb Power-of-Two Codes" *NASA Tech Briefs*, November 1, 2007: NPO-41336.
- [12] A. Kiely, M. Klimesh, "Fast Lossless Compression of Multispectral-Image Data" *NASA Tech Briefs*, July, 2002: NPO-21101.
- [13] M. Klimesh, "A Bit-Wise Adaptable Entropy Coding Technique" *NASA Tech Briefs*, August, 2006: NPO-42517.
- [14] N. Aranki, D. Keymeulen, A. Bakhshi and, M. Klimesh. "Fast and Adaptive Lossless on-board Hyperspectral Data Compression System for Space Applications", In *IEEE Aerospace Conference*. 9-13 March 2009.
- [15] M. Klimesh "Lossless, Multi-Spectral Data Compressor for Improved Compression for Pushbroom-Type Instruments", *NASA Tech Briefs*, July, 2008: NPO 45473.
- [16] J. Hoberock and N. Bell, "Thrust: A Parallel Template Library version 1.3.0", 2010, In <http://www.meganevtons.com/>.
- [17] AVIRIS, Hawaii, Scene 1, 2001, Flight f011020t01p03r05 <http://compression.jpl.nasa.gov/hyperspectral/>
- [18] 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.
- [19] *Lossless Multispectral & Hyperspectral Image Compression*. Draft Recommendation for Space Data System Standards, CCSDS 123.0-R-1. Red Book. Issue 1. Washington, D.C.: CCSDS, May 2011. (<http://public.ccsds.org/review>)
- [20] M. Klimesh, A. Kiely, P. Yeh, "Fast Lossless Compression of Multispectral and Hyperspectral Imagery," *Proc. International Workshop on On-Board Payload Data Compression (OBPDC)*, 8 pages, Toulouse, France, Oct. 28–29, 2010.

BIOGRAPHY



Nazeeh Aranki received the BSEE, MSEE and Ph.D. in Electrical Engineering from Caltech and USC. Nazeeh has 28 years of experience in design and implementation of digital and FPGA based systems. Since he joined JPL in 1994, his research interests have included reconfigurable hardware, digital signal and image processing, data compression, parallel processing, evolvable hardware, and neural networks. Nazeeh developed algorithms for compression of hyperspectral data and their implementations on reconfigurable platforms. He was awarded a patent and the NASA Space Act award for his contribution in the development of an FPGA-based neuroprocessor for automotive applications in control and diagnostics. He also served as the principal investigator and task Manager on a number of NASA, DARPA and AFRL projects related to data compression and power aware computing and communications.



Ben Hopson received a Master of Mathematics from Oxford University in 1998 and a BEng in Electronic Engineering from the University of Edinburgh in 2010. He is currently undertaking a PhD in Hardware / Algorithm Co-design as part of the Institute for Integrated Micro and Nano Systems at the University of Edinburgh, supervised by Dr Khaled Benkrid. He worked for several years for CESG involving cryptography, high performance computing, and algorithm design – and now consults for private companies in algorithm design applied to electronic engineering



Didier Keymeulen received the BSEE, MSEE and Ph.D. in Electrical Engineering and Computer Science from the Free University of Brussels, Belgium in 1994. In 1996 he joined the computer science division of the Japanese National Electrotechnical Laboratory as senior researcher. Currently he is principal member of the technical staff of JPL in the Bio-Inspired Technologies Group. At JPL, he is responsible for DoD and NASA applications on evolvable hardware for adaptive computing that leads to the development of fault-tolerant electronics and autonomous and adaptive sensor technology. He participated also as test electronics lead, to Tunable Laser Spectrum instrument on Mars Science Laboratory. He served as the chair, co-chair, and program-chair of the NASA/ESA Conference on Adaptive Hardware. Didier is a member of the IEEE.



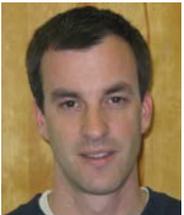
Khaled Benkrid is a Senior Lecturer in Electronic Engineering in the School of Engineering at the University of Edinburgh, Scotland, UK. During the last 13 years, he has been actively researching the areas of high performance computing using reconfigurable hardware and multi-core processors, and electronic design

automation. To date, his research in these areas has resulted in over 100 international journal and conference papers, with contributions in novel architectures, software tools, and applications in digital signal processing, communications, control systems, and scientific computing. Dr. Benkrid holds a PhD in Computer Science, a 1st Class “Ingénieur d’Etat” degree in Electronic Engineering, and an Executive MBA with Distinction. He is Senior IEEE Member and a Chartered UK Engineer.



Matthew Klimesh received B.S.E., M.S.E., and Ph.D. degrees, all in electrical engineering from the University of Michigan in Ann Arbor, in 1989, 1990, and 1995, respectively. He spent one year as a research fellow (postdoc) at Michigan. Since 1996 he has been with the Information Processing Group at Caltech’s Jet Propulsion Laboratory, working primarily

on research and development of data compression algorithms for space applications. His research interests include source coding, data compression, network coding, rate-distortion theory, channel coding, probability, and discrete mathematics.



Aaron Kiely received the BS, MS, MSE, and PhD degrees from Virginia Tech, the University of Southern California, the University of Michigan, and the University of Michigan, respectively. Since 1993, he has worked in the Information Processing Group of the Communications Architectures and Research Section at JPL, where he

conducts research in data compression and error-correcting codes. He wrote the emerging CCSDS 123.0 standard for Lossless Multispectral & Hyperspectral Image Compression and is the chair of the Multispectral and Hyperspectral Data Compression working group of CCSDS. Aaron led the development of the ICER image compression algorithm being used by the Mars Exploration Rovers, and he has also provided data compression consulting for several other deep space missions and instruments. Aaron has served on the faculty of Caltech, where he has periodically taught graduate-level courses on data compression and error-correcting codes.