UNIVERSIDADE D
COIMBRA

Óscar Almeida Ferraz

COMBINING LOW-POWER WITH PARALLEL PROCESSING FOR MULTISPECTRAL AND HYPERSPECTRAL IMAGE COMPRESSION

Master thesis submitted to the Integrated Master in Electrical and Computer Engineering, specialization in computers, supervised by Professor Gabriel Falcão Paiva Fernandes and cosupervised by Professor Vitor Manuel Mendes da Silva, and presented to the Department of Electrical and Computer Engineering of the Faculty of Science and Technology of the University of Coimbra.

September 2019

Combining Low-Power With Parallel Processing for Multispectral and Hyperspectral Image Compression

Óscar Almeida Ferraz

Dissertation submitted to the Department of Electrical and Computer Engineering of the Faculty of Science and Technology of the University of Coimbra to obtain the Master's Degree in

Electrical and Computer Engineering

Supervisor:	Gabriel Falcão Paiva Fernandes, PhD
Cosupevisor:	Vitor Manuel Mendes da Silva, PhD

Jury

President:	Luís Alberto da Silva Cruz, PhD
Supervisor:	Gabriel Falcão Paiva Fernandes, PhD
Member:	Marco Alexandre Cravo Gomes, PhD

September 2019

A bit beyond perception's reach
I sometimes believe I see
That life is two locked boxes, each
Containing the other's key.

- Piet Hien

Agradecimentos

Em primeiro lugar, gostaria de prestar um especial agradecimento aos meus orientadores Doutor Gabriel Falcão e Doutor Vitor Silva pela mentoria, conhecimento, paciência, motivação e sobretudo pelo profissionalismo e pela constante disponibilidade e ajuda que mostraram ao longo da realização desta dissertação. Estou muito agradecido.

Em segundo lugar, gostava de agradecer ao instituto de telecomunicações de Coimbra pelo espaço e material cedido durante o desenvolvimento da dissetação. Também quero agradecer ao pessoal do IT, em especial ao Manel, ao Thyago e ao professor Marco pelos conselhos dados para a realização da dissertação. E também sem esquecer o César, a Andreia, o Pedro, o Filipe e o João e toda a outra malta do IT. Obrigado pela Companhia!

Aos meus pais pela educação que deram, por me ensinarem que é preciso trabalhar para alcançar coisas na vida, pela paciência que tiveram, por nunca terem desistido de mim e, sobretudo, pelo grande esforço que fizeram para conseguirem manter-me na universidade. Esta tese é dedicada a vocês e estarei eternamente grato.

À minha madrinha Dilma pelo impacto que tens na minha vida, e por me teres guiado nos momentos mais difíceis.

Aos meus avós pela educação que me deram e a toda a minha família, em especial ao meu irmão, pela companhia desde da minha infância.

Aos meus amigos, Julien, Rui e o Ricardo pela amizade que se mantém desde dos tempos da primária. A todos os meus amigos e colegas de faculdade, Gabriel, Pedro, Shon, Freire, Seco, Daniel, Adriano e todos aqueles que viveram connosco o espírito boémio de Coimbra, mas também pelas noites de desabafo e frustração.

Ao Pedro Cunha, Paulo e Carlos pelos tempos de descontração passados na varanda durante as pausas na escrita da dissertação.

A todos a aqueles com quem me cruzei e que contribuíram para a minha formação como pessoa. Obrigado pelos bons tempos.

Por fim, um grande obrigado à Valéria, por tudo o que representas na minha vida e por todo o apoio, preocupação e paciência que tens comigo. Obrigado por me tornares numa pessoa melhor.

Obrigado a todos!

Abstract

The consultative committee for space data systems (CCSDS) 123 is a hyperspectral and multispectral image compression algorithm composed of a predictor and an encoder. Usually, the systems that generate these types of images (satellites, drones, etc.) have energy restrictions. Hence, field-programmable gate arrays (FPGAs) show themselves as efficient devices to implement the CCSDS 123 due to its low energy consumption. The smartphone market has turned central processing units (CPUs) and graphics processing units (GPUs) into energy-efficient systems, making them potential competitors against FPGAs implementation dominance in the field of low-energy compression.

The objective of this dissertation is, using a low-power GPU (Jetson TX2), to parallelize the CCSDS 123. Intra-band prediction ($P = 0$) uses a single kernel. When using inter-band prediction ($P > 0$), the predictor has data dependencies within bands, making parallelization less efficient and more challenging to implement. Hybrid parallelizations (CPU+GPU) are studied for the two encoders designed for this standard, producing a heterogeneous computing system.

The implementations are subject to tests that compare the parallel execution times with the serial execution times in order to identify the best implementations. An energy analysis is performed, measuring the power used by the board over the algorithm's running time. In the end, the throughput rate and energy efficiency are compared with the state-of-the-art.

Keywords

CCSDS 123, Low-Power GPUs, CUDA, Parallel Programming, Hyperspectral and Multispectral Image Compression, Parallel Processing, Lossless Compression

Resumo

O consultative committee for space data systems (CCSDS) 123 é um algoritmo de compressão de imagens hiperespectrais e multiespectrais composto por um preditor e um codificador. Normalmente, os sistemas que geram este tipo de imagens (satélites, drones, etc...) têm restrições energéticas. Este algoritmo é implementado, sobretudo em field-programmable gate arrays (FPGAs) devido ao seu baixo consumo energético. O mercado dos smartphones tem tornado os central processing units (CPUs) e graphics processing units (GPUs) em dispositivos energeticamente eficientes, colocando-os em posição de competir contra as FPGAs no campo de compressão de baixo consumo.

O objetivo desta dissertação é, utilizando uma Jetson TX2, paralelizar o CCSDS 123. No preditor, quando a predição é intra-banda ($P = 0$), é utilizado um único kernel. Quando se usa predição inter-banda ($P > 0$), o preditor passa a ter dependências de dados dentro das bandas, tornando a paralelização menos eficiente e mais difícil de implementar. No codificador, que contém dependências de dados, são estudadas paralelizações utilizando vários dispositivos (CPU+GPU) nos dois codificadores contemplados nesta norma. Produzindo uma solução híbrida de computação heterogénea.

As implementações são alvo de testes que compararam o tempo de execução paralela com os tempos execução em série de forma a identificar as melhores implementações. Ainda é feita uma análise energética medindo a potência utilizada pela placa ao longo do tempo de execução do algoritmo. No final, a taxa de débito e a eficiência energética são comparadas com o estado de arte.

Palavras Chave

CCSDS 123, GPUs de Baixo Consumo, CUDA, Programação Paralela, Compressão de Imagens Hiperspectrais e Multiespectrais, Compessão sem Perdas, Processamento Paralelo

Contents

List of Figures

List of Tables

List of Acronyms

ALU arithmetic logic unit

API application programming interface

AVIRIS airborne visible/infrared imaging spectrometer

BIL band-interleaved-by-line order

BIP band-interleaved-by-pixel order

BSQ band-sequential order

CASI compact airborne spectrographic imager

CRISM compact reconnaissance imaging spectrometer for mars

CCSDS consultative committee for space data systems

CDS coded data set

CPU central processing unit

CUDA compute unified device architecture

DtoH device to host

DSP digital signal processor

ESA european space agency

FLOPS floating point operations per second

FPGA field-programmable gate array

FS fundamental sequence

HtoD host to device

GPU graphics processing unit

GPGPU general purpose graphics processing unit

LPDDR4 low power double data rate 4

LWIR long wave infrared

LSB least significant bit

MHI multispectral and hyperspectral image

MPR mapped prediction residual

OpenACC open accelerators

OpenCL open computing language

OpenMP open multi-processing

OS operative system

POSIX portable operating system interface

PWM pulse width modulation

RAM random access memory

RTL register transfer level

ROS remainder-of-segment

SIMT single instruction, multiple thread

SIMD single instruction, multiple data

SM streaming multiprocessor

SoC system on a chip

SWIR short wave infrared

VHDL very high speed integrated circuits hardware description language

1

Introduction

Contents

A spectral image is taken across multiple bands through the electromagnetic spectrum. While an ordinary camera captures light in the visible spectrum, spectral images may use infrared, visible, ultraviolet, x-rays, or some combination of those spectrums. As a consequence of high number of spectrums used on these images, spectral imaging can be divided into 2 groups: multispectral and hyperspectral imaging. These types of imaging are used in various scientific areas, such as astronomy, agriculture, physics, geosciences and others.

Multispectral and hyperspectral images (MHIs) acquired in systems such as satellites or aircrafts, must be transmitted to earth-based stations for analysis. As the sensor's resolution increases, the collected information also increases [11]. Hence, compression mechanisms are applied to decrease memory storage and transmission bandwidth by reducing data volume [5, 12, 13].

In order to develop standards in communication and data systems for space applications, the consultative committee for space data systems (CCSDS) was founded in 1982, constituted by multiple space agencies that meet periodically to establish norms [5]. One of the standard algorithms proposed for lossless compression of MHIs is the CCSDS 123 [5]. This standard is composed by a predictor and an encoder.

The predictor uses a low complexity adaptive linear prediction model predicting a value based on a three-dimensional neighborhood, which outputs a mapped residual. The residual is the difference between the predicted value and the actual sample [14]. Of all configurable predictors parameters that affect performance and compression ratios, the most important are the number of bands used in the neighborhood, the in-band type of neighborhood (column-oriented or neighborhood-oriented) and the prediction mode (full mode or reduced mode) [14, 15]. Subsection 2.2.1 describes the predictor in more detail.

The encoder produces a compressed image containing a header with metadata and a body with the encoded mapped prediction residuals (MPRs). The residuals can be encoded using two modes: the sample-adaptive entropy encoder uses Golomb-power-of-2 codes which outputs variable-length binary codewords that adapt to image statistics. The block-adaptive entropy encoder is the CCSDS 121, an older standard, which uses Rice coding. The CCSDS 121 can be used as an encoder in the newer standard to take advantage of previous developed space-qualified hardware [14]. This coder packs the MPRs into blocks and applies different independent methods to select the shortest codeword. Further details are explained in subsection 2.2.2.

In Figure 1.1 is represented the basic flowchart of the CCSDS 123. The predictor receives input samples and outputs MPRs to be encoded by the entropy encoder. From [1], a serial version of the algorithm achieved 45% of processing time for the predictor and 55% for the encoder.

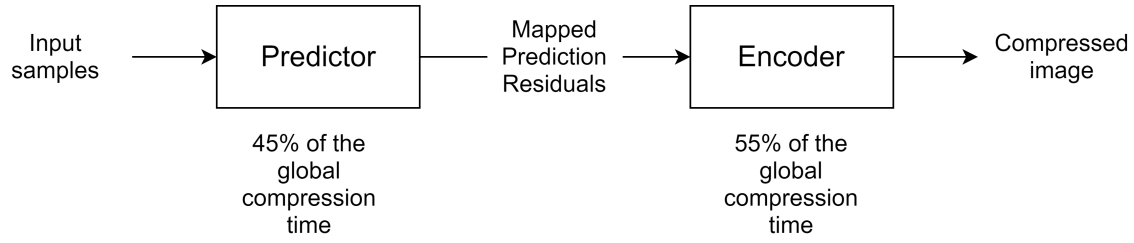


Figure 1.1: Rate of processing times for compression of MHIs on CPU [1].

Satellites and some aircrafts that employ this standard, impose several limitations in terms of space, weight, power requirements and energy consumption. Therefore, solutions must meet those requirements while maintaining high throughput. In recent years, graphics processing units (GPUs) have shown to be an effective way of attaining high throughput performances with low development efforts compared to CPUs and field-programmable gate arrays (FPGAs) [9].

1.1 Motivation

Distinctively from conventional systems processing on earth-based stations with small or no power constraints, on-board systems present energy requirements and compression algorithms need to be tailored to execute on these devices.

GPUs are gaining relevance in several domains besides rendering graphic applications. Recently, the smartphone market is turning CPUs and GPUs into more energy-efficient devices, making them promising candidates for MHI compression in on-board systems such as satellites and aircrafts. While the usual power required to run a desktop GPU is around 250~300 W, current GPU families incorporate low-power devices ($<$15 W), delivering hundreds of cores operating at ~1 GHz of clock frequency, supporting multithreading, single instruction, multiple thread (SIMT) and single instruction, multiple data (SIMD) execution models and parallel programming frameworks as, for instance, compute unified device architecture (CUDA) and open accelerators (OpenACC) [7].

For various years, the MHI compression field is dominated by FPGAs. Little research has been done using alternative implementations. These new types of low-power GPUs can present a new solution that can benefit in terms of throughput performance, efficiency, cost and development effort to the traditional implementations of FPGAs.

1.2 Objectives

This thesis proposes an alternative to the dominating approach of using FPGAs on MHI compression. This document analyzes an alternative solution exploiting the through-

put performance and energy efficiency of low-power GPU, namely, the Nvidia Tegra TX2 GPU, and compare it with state-of-the-art solutions for the CCSDS 123 standard. Therefore, the main objectives of this thesis are:

- Investigate the state-of-the-art developments of the CCSDS 123 algorithm [5, 14];
- Develop parallel functions to parallelize the predictor in CUDA and both encoders;
- Explore CUDA optimization techniques, such as, the use of shared, texture and constant memory, memory coalescence, optimal thread occupancy and high arithmetic intensity;
- Assess the viability of using heterogeneous computation, running the predictor on GPU and the encoder on CPU as other combinations;
- Tune the parameters of the algorithm for faster execution times and compression rates;
- Measure execution times and energy consumption of the algorithm;
- Evaluate throughput and energy efficiency of the low-power proposed solution and analyze it against state-of-the-art approaches;

1.3 Main contributions

Using parallel computing in signal processing can offer advantages in both time and energy efficiency. The results are thought to be particularly good for the computation of residuals, reassuring that the performance (throughput / energy) bridge is closing between FPGAs and GPUs.

This dissertation offers and heterogeneous implementation of the CCSDS 123. The predictor is implemented using CUDA in two distinct ways: inter-band prediction ($P > 0$) and intra-band prediction ($P = 0$), which achieves a throughput performance of 1.031 GSa/s. This work also offers the parallel implementations for both sample and block adaptive encoders. One version of the sample adaptive encoder is implemented in CUDA, and another using all the CPU cores. The block adaptive encoder uses a combination of GPU+CPU. Lastly, it is provided an energy analysis using the 2 power modes implemented in the Jetson TX2.

This work resulted in an article entitled ”Combining Low-Power With Parallel Processing for Multispectral and Hyperspectral Image Compression Through Roofline Model Analysis”, submitted in September 2019 to the IEEE International Conference on Acoustics, Speech, and Signal Processing 2020 (ICASSP).

1.4 Dissertation Outline

This thesis is structured in 7 chapters. It starts with a short introduction followed by the principles of multispectral and hyperspectral imaging and its sensors as well as the technical description of the CCSDS 123 algorithm in chapter 2. In chapter 3, it is presented the Nvidia's Jetson TX2 hardware description and a brief introduction to CUDA. Next, in chapter 4, it is analyzed multiple state-of-the-art implementations of the CCSDS 123 . Chapter 5 presents a discussion about the parallelization and optimization techniques used in this thesis. The next chapter exposes the obtained results and its energy analysis. In the final chapter, a conclusion is performed from the results and future work lines, goals and possibilities are discussed.

2

Multispectral and Hyperspectral Lossless Compression

Contents

This chapter depicts an overview of multispectral and hyperspectral differences and shows examples of 2 types of sensors. Further on, is shown a description in detail of the predictor and encoders for the lossless compression algorithm proposed by the consultative committee for space data systems (CCSDS).

2.1 Multispectral and Hyperspectral Imaging

The main difference between multispectral images and hyperspectral images is the number of bands in the spectral domain and the wavelength range of those bands. Multispectral images typically incorporate between 3 to 10 bands [16, 17]. For example, the Landsat-8 is a multispectral sensor capable of detecting wavelengths between 400 *nm* and 12500 *nm* in 11 bands [16]. An example of a hyperspectral sensor is the airborne visible/infrared imaging spectrometer (AVIRIS). It delivers 224 contiguous spectral channels with wavelengths ranging from 400 *nm* and 2500 *nm* [16, 18]. Accordingly to [16], for multispectral sensors, for each band the wavelength ranges approximately 100 *nm* while bands of hyperspectral images have a narrower wavelength on each band, between 10 *nm* and 20 *nm*. As an analogy of the paragraph above, Figure 2.1 illustrates the differences between multispectral and hyperspectral images.

A representation of a hyperspectral image can be seen in Figure 2.2. The cube front plane (XY) represents the spatial coordinates in the same spectral domain, while the coordinate Z represents the variation on the spectral domain. The color variations on ZX and ZY planes are simply a representation of sample values.

2.2 CCSDS 123 Standard

The CCSDS proposed a solution for the lossless compression of multispectral and hyperspectral images (MHIs) [5, 14]. The CCSDS 123 is composed of two main parts: a predictor and an encoder as shown in Figure 1.1. The predictor is fed with a 3D image with a size N_x by N_y and N_z which represents the number of bands.

Since the MHI has 3 dimensions, the image can be scanned in many forms. The algorithm contemplates 3 ways: in band-sequential order (BSQ), the samples are read in the same band, line by line, in band-interleaved-by-pixel order (BIP), the image is read on the same pixel, band by band and band-interleaved-by-line order (BIL), which reads lines, band by band as shown in Figure 2.3. These scanning methods are also applied to encode the mapped prediction residuals (MPRs) after the prediction stage. All symbols for this algorithm can be consulted in Tables A.1, A.2 and A.3 on appendix A

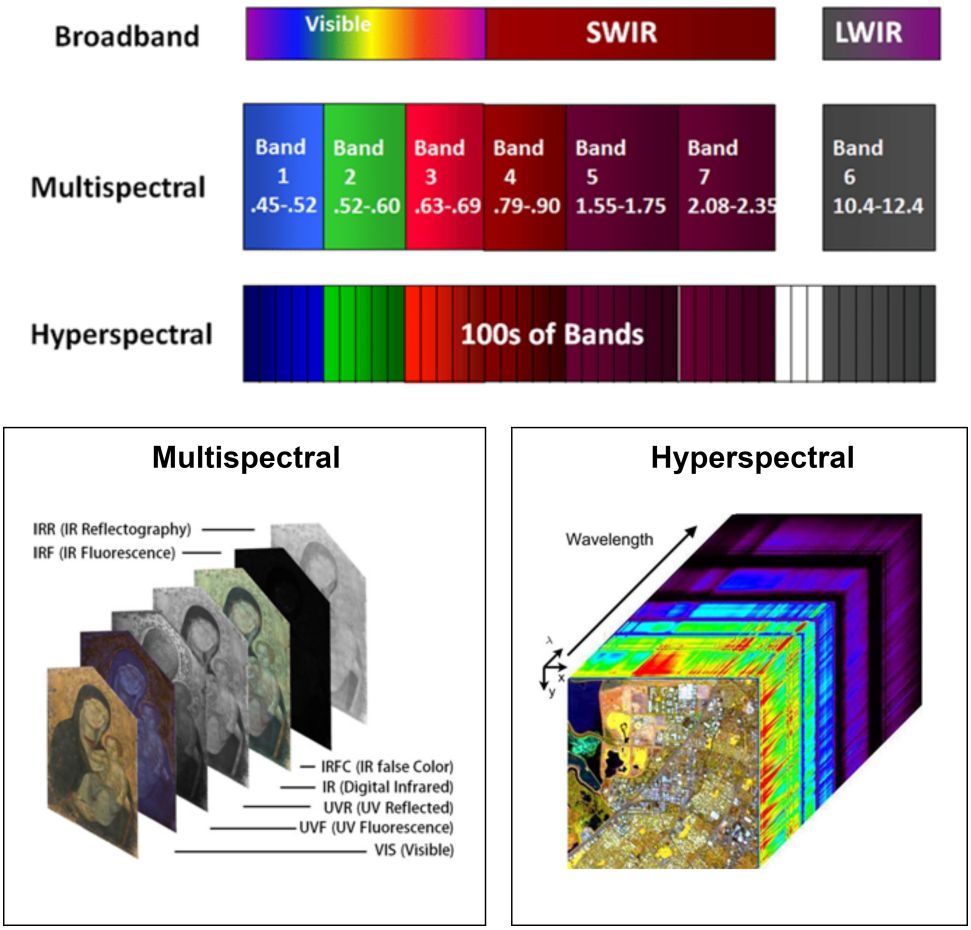


Figure 2.1: Differences between multispectral and hyperspectral images in visible, short wave infrared (SWIR) and long wave infrared (LWIR) spectrums, extracted from [2].

2.2.1 Predictor

The first part of the CCSDS 123 uses an adaptive linear prediction model. The purpose of this component is to calculate the difference between the observed value and the predicted value, outputting residuals with low entropy. The encoder codifies those residuals and will result in a shorter codeword compared to encoding raw samples. Figure 2.4 shows the flowchart for the predictor of the CCSDS 123 algorithm. Variables in bold represent vectors.

The predictor outputs mapped prediction residuals from input samples $s_{z,y,x}$ (or $s_z(t)$. t is the alternative indexing variable where:

$$t = y \cdot N_x + x. \tag{2.1}$$

The local sum $\sigma_{z,y,x}$ is calculated from the input samples, where two modes can be used:

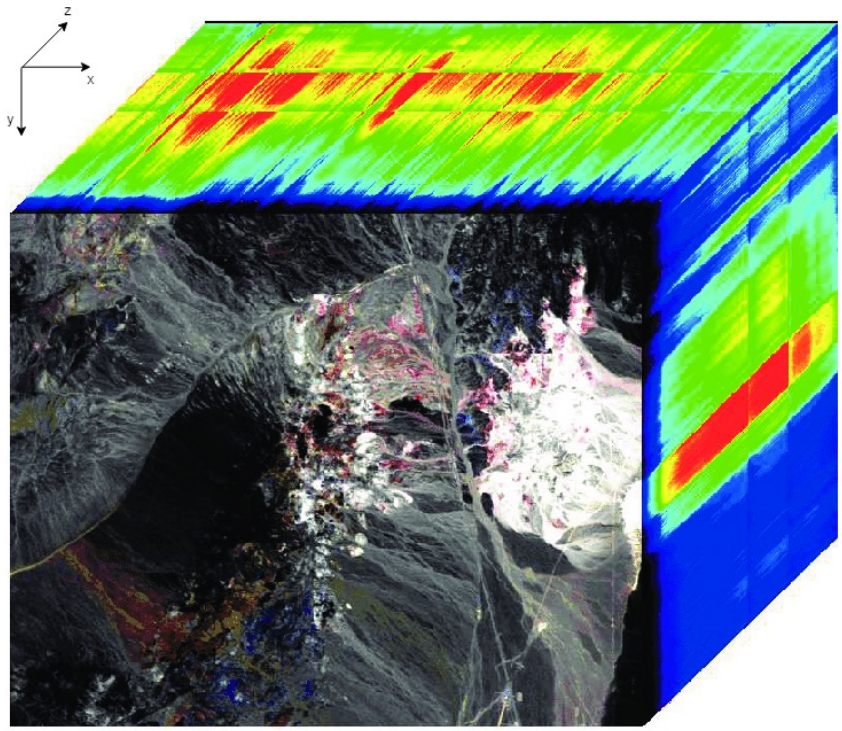


Figure 2.2: Hyperspectral image with 224 bands from Cuprite Mining District, Nevada, courtesy of [3].

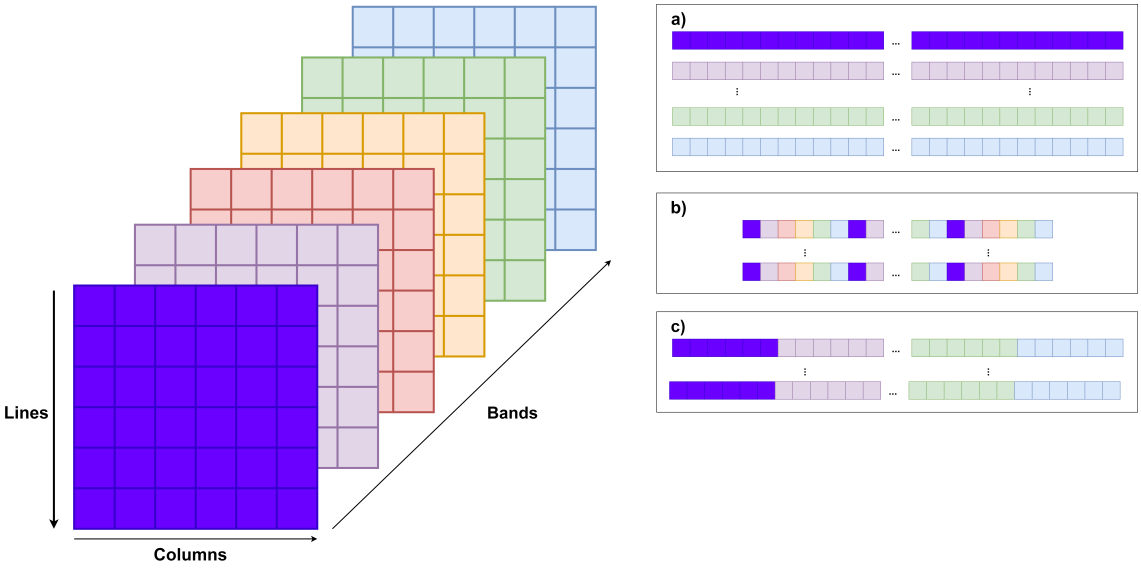


Figure 2.3: Reading and encoding orders, courtesy of [4]. a) Band sequential order (BSQ), b) Band interleaved by pixel (BIP), c) Band interleaved by line (BIL).

the neighbor-oriented sum uses the neighbor values from the same band as the input sample, as depicted in Figure 2.5. The column-oriented sum calculates the sum from the neighbor pixel in the same column and band. (2.2) and (2.3) describe the formula to calculate the sum by the neighbor-oriented and column-oriented methods respectively [5].

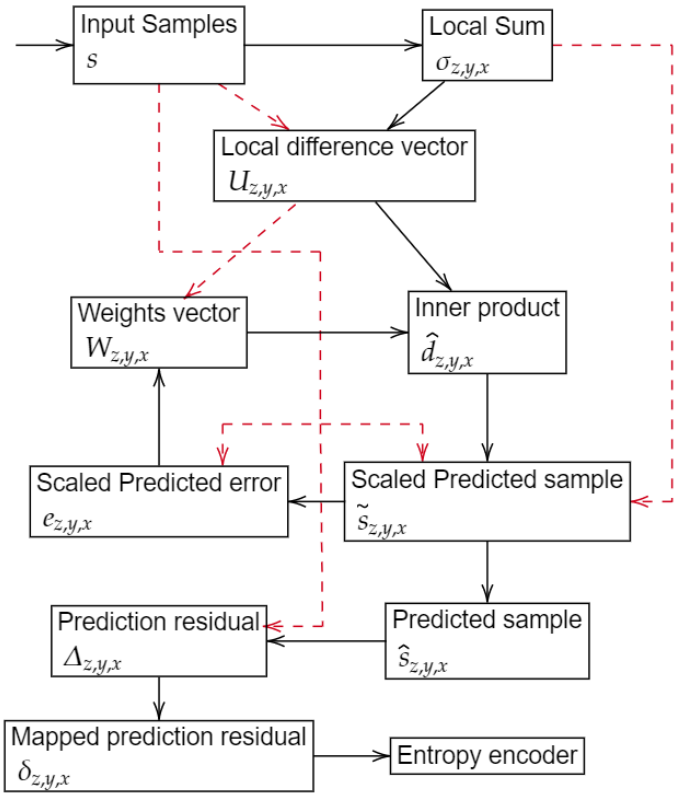


Figure 2.4: CCSDS 123 predictor algorithm flowchart. The dashed red arrows point to blocks where previous values are needed for the calculation. The black arrow shows the main computation flow.

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}, & y > 0,\ 0 < x < N_x - 1, \\ 4s_{z,y,x-1}, & y = 0,\ x > 0, \\ 2(s_{z,y-1,x} + s_{z,y-1,x+1}), & y > 0,\ x = 0, \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x}, & y > 0,\ x = N_x - 1, \end{cases} \tag{2.2}$$

$$\sigma_{z,y,x} = \begin{cases} 4s_{z,y-1,x}, & y > 0, \\ 4s_{z,y,x-1}, & y = 0,\ x > 0, \\ \textit{not needed}, & y = 0,\ x = 0. \end{cases} \tag{2.3}$$

The central local difference $d_{z,y,x}$ is calculated when x and y are both different from zero

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x}, \tag{2.4}$$

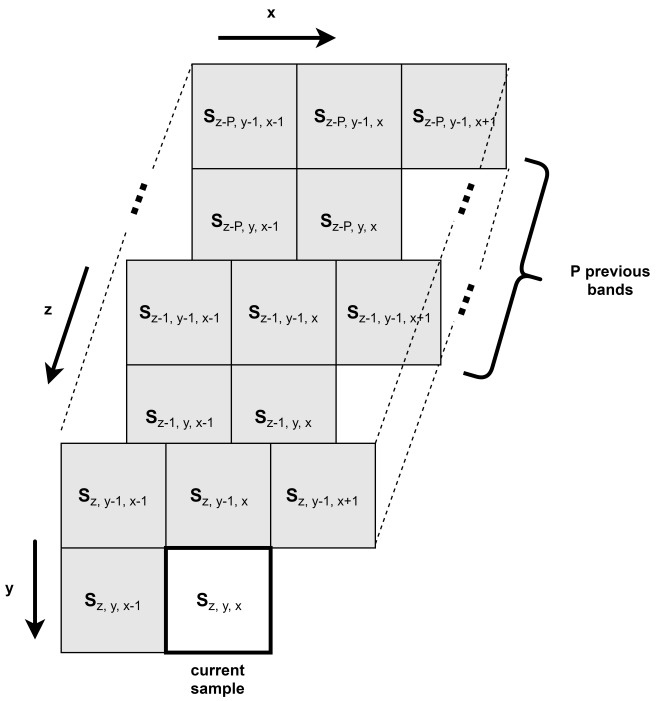


Figure 2.5: Typical Prediction Neighborhood, courtesy of [5].

analogously to the (2.4), when x and y are not both zero, the directional local differences are defined as:

$$d^N_{z,y,x} = \begin{cases} 4s_{z,y-1,x} - \sigma_{z,y,x}, & y > 0, \\ 0, & y = 0, \end{cases} \tag{2.5}$$

$$d^W_{z,y,x} = \begin{cases} 4s_{z,y,x-1} - \sigma_{z,y,x}, & x > 0,\ y > 0, \\ 4s_{z,y-1,x} - \sigma_{z,y,x}, & x = 0,\ y > 0, \\ 0, & y = 0, \end{cases} \tag{2.6}$$

$$d^{NW}_{z,y,x} = \begin{cases} 4s_{z,y-1,x-1} - \sigma_{z,y,x}, & x > 0,\ y > 0, \\ 4s_{z,y-1,x} - \sigma_{z,y,x}, & x = 0,\ y > 0, \\ 0, & y = 0, \end{cases} \tag{2.7}$$

where $d^N_{z,y,x}$ and $d^W_{z,y,x}$ are the difference to the upper and left sample respectively, and $d^{NW}_{z,y,x}$ is the difference to the upper-left sample.

The central local difference $d_{z,y,x}$ and the the directional local differences $d^N_{z,y,x}$, $d^W_{z,y,x}$ and $d^{NW}_{z,y,x}$ are used to create a local difference vector $\boldsymbol{U_z}(t)$. The vector is constructed differently in the two prediction modes. On full prediction mode it is defined as:

$$U_z(t) = \begin{bmatrix} d_z^N(t) \\ d_z^W(t) \\ d_z^{NW}(t) \\ d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P}(t) \end{bmatrix}. \tag{2.8}$$

P must be a value between 0 and 15 and is the number of spectral bands used for predicting the current band. Under the reduced prediction mode, $\boldsymbol{U_z}(t)$ is defined as:

$$U_z(t) = \begin{bmatrix} d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P}(t) \end{bmatrix}. \tag{2.9}$$

As shown in (2.9), the reduced mode ignores the directional local differences $d_{z,y,x}^N$, $d_{z,y,x}^W$ and $d_{z,y,x}^{NW}$.

A predicted central local difference $\widehat{d_z}(t)$ is the inner product of vector $\boldsymbol{U_z}(t)$ and vector of weights $\boldsymbol{W_z}(t)$. This vector $\boldsymbol{W_z}(t)$ decides the difference that is making a better prediction of the trend in sample values. For the full prediction mode:

$$W_z(t) = \begin{bmatrix} \omega_z^N(t) \\ \omega_z^W(t) \\ \omega_z^{NW}(t) \\ \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \vdots \\ \omega_z^{(z-P)}(t) \end{bmatrix}, \tag{2.10}$$

and for the reduced prediction mode:

$$W_z(t) = \begin{bmatrix} \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \vdots \\ \omega_z^{(z-P)}(t) \end{bmatrix}. \tag{2.11}$$

For each band the initial weight vector $\boldsymbol{W_z}(1)$ may be initialized in 2 modes: default mode and custom mode. In default mode, the weight components $\omega_z^{(i)}(t)$ are derived from:

$$\begin{gathered}\omega_z^{(1)}(1) = \tfrac{7}{8}2^{\Omega}, \\ \omega_z^{(i)}(1) = \left\lfloor \tfrac{1}{8}\omega_z^{(i-1)}(1) \right\rfloor, \quad i = 2,3,...,z-P, \\ \omega_z^{N}(1) = \omega_z^{W}(1) = \omega_z^{NW}(1) = 0,\end{gathered} \tag{2.12}$$

being Ω the resolution of the weight values, this value is user-defined and must lie between 4 and 19.

In the custom weight initialization mode, the user specifies a custom weight initialization vector $\boldsymbol{\Lambda_z}$. This vector may be chosen according to the instrument characteristics, training data or a vector from a previous compressed image. The vector is calculated from:

$$W_z(1) = 2^{\Omega+3-Q}\Lambda_z + \left\lceil 2^{\Omega+2-Q} - 1 \right\rceil \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}. \tag{2.13}$$

Q is the weight initialization resolution and is a user specified-parameter between 3 and Ω+3.

The predicted central local difference $\widehat{d}_z(t)$ is the inner product of weight vector $\boldsymbol{W_z}(t)$ and local difference vector $\boldsymbol{U_z}(t)$:

$$\widehat{d}_z(t) = W_z^{\top}(t)U_z(t). \tag{2.14}$$

These equations are used to calculate a scaled predicted sample value defined as:

$$\widetilde{s}_z(t) = \begin{cases} clip\left(\left\lfloor \frac{mod_R^*[\widehat{d}_z(t) + 2^{\Omega}(\sigma_z(t) - 4s_{mid})]}{2^{\Omega+1}} \right\rfloor + 2s_{mid} + 1, \{2s_{min}, 2s_{max}+1\} \right), & t > 0, \\ 2s_{z-1}(t), & t = 0,\ P > 0,\ z > 0, \\ 2s_{mid}, & t = 0 \text{ and } (P > 0 \text{ or } z > 0), \end{cases} \tag{2.15}$$

where s_{max}, s_{min} and s_{mid} are the upper limit, lower limit and mid-range value, respectively, of data sample's dynamic range D. This dynamic range must be a value between 2 and 16 bits:

$$\begin{cases} s_{min} = 0,\ s_{max} = 2^D - 1,\ s_{mid} = 2^{D-1}, & \textit{unsigned integers}, \\ s_{min} = -2^{D-1},\ s_{max} = 2^{D-1} - 1,\ s_{mid} = 0, & \textit{signed integers}. \end{cases} \tag{2.16}$$

R is a user selected size and must be a value in the range of $max\{32, D+\Omega+2\} \leq R \leq 64$. The function $clip$ denotes the clipping of the number into the range designated in brackets, in this case the range is $\{2s_{min}, 2s_{max}+1\}$.

The function mod_R^* is defined as:

$$mod_R^*[x] = \left(\left(x+2^{R-1}\right) mod\ 2^R\right) - 2^{R-1}, \tag{2.17}$$

where:

$$M\ mod\ n = M - n\left\lfloor \frac{M}{n} \right\rfloor. \tag{2.18}$$

$mod_R^*[x]$ is a natural result of storing a signed integer x in a R-bit register in two's complement form.

The predicted sample value $\widehat{s}_z(t)$ is then calculated:

$$\widehat{s}_z(t) = \left\lfloor \frac{\widetilde{s}_z(t)}{2} \right\rfloor, \tag{2.19}$$

and used by the prediction residual $\Delta_z(t)$:

$$\Delta_z(t) = s_z(t) - \widehat{s}_z(t), \tag{2.20}$$

and the difference to the nearest end-point $\theta_z(t)$:

$$\theta_z(t) = min\left\{\widehat{s}_z(t) - s_{min}, s_{max} - \widehat{s}_z(t)\right\}, \tag{2.21}$$

to finally calculate the mapped prediction residuals $\delta_z(t)$:

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t), & |\Delta_z(t)| > \theta_z(t), \\ 2\,|\Delta_z(t)|, & 0 \leq (-1)^{\widetilde{s}_z(t)}\Delta_z(t) \leq \theta_z(t), \\ 2\,|\Delta_z(t)| - 1, & otherwise. \end{cases} \tag{2.22}$$

The weight vector $\boldsymbol{W_z}(t)$ is updated every iteration. First the scaled prediction error $e_z(t)$ is calculated:

$$e_z(t) = 2s_z(t) - \widetilde{s}_z(t), \tag{2.23}$$

then the weight update scaling exponent $\rho(t)$ is calculated:

$$\rho(t) = clip\left(v_{min} + \left\lfloor \frac{t - N_x}{t_{inc}} \right\rfloor, \{v_{min}, v_{max}\}\right) + D - \Omega, \tag{2.24}$$

v_{min} and v_{max} are user defined integers satisfying the following condition $-6 \leq v_{min} \leq v_{max} \leq 9$. The weight update factor t_{inc} is a power of 2 between 2^4 and 2^{11}. These parameters control the rate of the weight adaptation to image statistics. Smaller values of $\rho(t)$ produce a faster adaptation but worse compression performance [5].

Finally, the weight vector is updated:

$$W_z(t+1) = clip\left(W_z(t) + \left\lfloor \frac{1}{2}\left(sgn^+[e_z(t)] \cdot 2^{-\rho(t)} \cdot U_z(t) + 1\right)\right\rfloor, \{\omega_{min}, \omega_{max}\}\right), \tag{2.25}$$

where the function $sgn^+(x)$ returns 1 if $x \geq 0$ and returns -1 if $x < 0$. ω_{min} and ω_{max} are the minimum ($-2^{\Omega+2}$) and maximum ($-2^{\Omega+2} - 1$) weight values respectively.

2.2.2 Encoder

The encoder produces a compressed image, which is composed of a variable-length header and a body containing the encoded residuals. The header contains the image, predictor and coder metadata, while the body contains the encoded MPRs $\delta_z(t)$, as shown in Figure 2.6. The fields of these headers are described in tables B.1, B.2, B.3 and B.4 from appendix B.

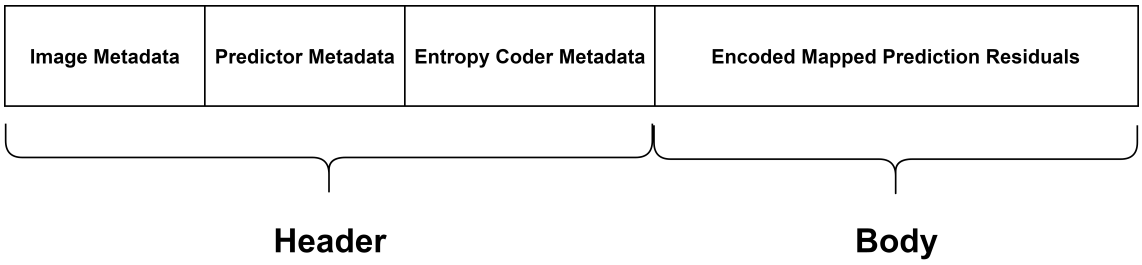


Figure 2.6: Compressed image structure.

In some cases, the MPR is encoded as the binary representation, however, in most cases the MPRs are encoded following the fundamental sequence (FS), also known as the alternative unary coding. For instance, the FS codeword of an arbitrary value n is composed of n 'zeros' followed by a 'one' as shown by Figure 2.7.

Those MPRs $\delta_z(t)$, can be encoded by 2 types of entropy coders: the sample-adaptive entropy coder or the block-adaptive entropy coder.

Mapped Prediction residuals δ_i	FS Codeword
0	1
1	01
2	001
3	0001
...	...
2^{D-1}	0000000...00001 (2^{D-1} zeros)

Figure 2.7: Fundamental sequence codewords as a function of the MPR values δ_i, adapted from [6].

2.2.2.A Sample-Adaptive Entropy Coder

This coder uses Golomb-Power-of 2 coding to encode the MPRs $\delta_z(t)$ [19]. This method uses an accumulator $\Sigma_z(t)$ and a counter $\Gamma(t)$ that are adaptively updated during the encoding process to generate the compressed stream. These variables are initialized:

$$\Gamma_z(1) = 2^{\gamma_0}, \tag{2.26}$$

$$\Sigma_z(1) = \left\lfloor \frac{1}{2^7} \left(3 \cdot 2^{k_z'+6} - 49\right) \Gamma(1) \right\rfloor. \tag{2.27}$$

The initial count exponent γ_0 is a user defined integer between 1 and 8 and k_z' is also a user defined integer with range between zero and $D-2$. After the initialization, the variables are defined as:

$$\Sigma_z(t) = \begin{cases} \Sigma_z(t-1) + \delta_z(t-1), & \Gamma(t-1) < 2^{\gamma^*} - 1, \\ \left\lfloor \frac{\Sigma_z(t-1)+\delta_z(t-1)+1}{2} \right\rfloor, & \Gamma_z(t-1) = 2^{\gamma^*} - 1, \end{cases} \tag{2.28}$$

$$\Gamma_z(t) = \begin{cases} \Gamma_z(t-1) + 1, & \Gamma_z(t-1) < 2^{\gamma^*} - 1, \\ \left\lfloor \frac{\Gamma_z(t-1)+1}{2} \right\rfloor, & \Gamma_z(t-1) = 2^{\gamma^*} - 1. \end{cases} \tag{2.29}$$

The user defined rescaling counter size γ^* must be an integer between the range of $max\{4, \gamma_0 + 1\} \leq \gamma^* \leq 9$. This value controls the rate at which the counter $\Gamma(t)$ and the accumulator $\Sigma_z(t)$ are rescaled. Lower values of γ^* and γ_0 produce a faster adaptation to the image statistics, but worsen the steady-state performance [14].

To encode the first MPR of each band, the codeword is the D-bit unsigned binary integer representation of $\delta_z(0)$. For the other residuals, the codeword depends on the unary codeword length $u_z(t)$ and the variable length code parameter $k_z(t)$, where:

$$k_z(t) = \begin{cases} 0, & log_2\left(\frac{\Sigma_z(t)+\left\lfloor\frac{49}{128}\Gamma(t)\right\rfloor}{\Gamma(t)}\right) < 0, \\ D-2, & log_2\left(\frac{\Sigma_z(t)+\left\lfloor\frac{49}{128}\Gamma(t)\right\rfloor}{\Gamma(t)}\right) > D-2, \\ log_2\left(\frac{\Sigma_z(t)+\left\lfloor\frac{49}{128}\Gamma(t)\right\rfloor}{\Gamma(t)}\right), & otherwise, \end{cases} \tag{2.30}$$

and $u_z(t)$ is defined as:

$$u_z(t) = \left\lfloor \frac{\delta_z(t)}{2^{k_z(t)}} \right\rfloor. \tag{2.31}$$

U_{max} is a user supplied variable between 8 and 32. If $u_z(t) < U_{max}$ the codeword consists of $u_z(t)$ 'zeros' followed by a 'one' and $k_z(t)$ least significant bits of the binary representation of $\delta_z(t)$ as shown in Figure 2.8.

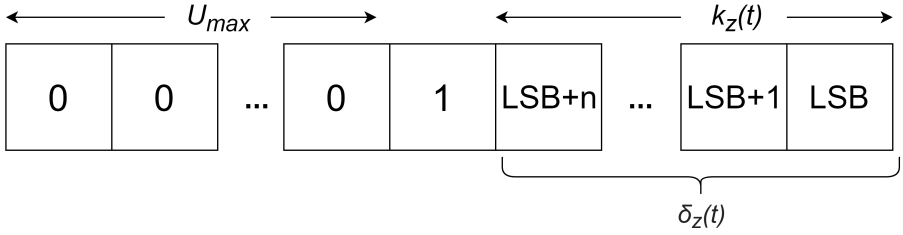


Figure 2.8: Codeword format for $u_z(t) < U_{max}$.

If the previous condition is not verified, the codeword consists of U_{max} 'zeros' and the binary representation of $\delta_z(t)$, as depicted in Figure 2.9.

When the encoding process is finished, zero padding must be done so the compressed image is multiple of the output word size B, which is a user-selected integer between 1 and 8.

2.2.2.B Block-Adaptive Entropy Coder

Another method to encode the MPRs $\delta_z(t)$ is the CCSDS 121, a 1D universal lossless encoder that uses Rice codes. The residuals are divided in blocks containing J residuals.

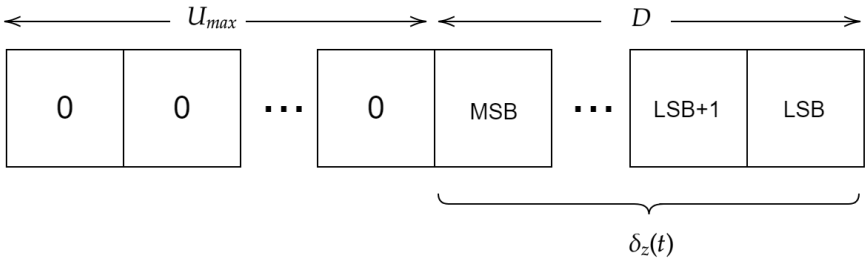


Figure 2.9: Codeword format for $u_z(t) \geq U_{max}$.

J is a user defined number that can be 8, 16, 32 or 64. Four algorithms are executed concurrently for each block resulting in 4 codewords. The shortest codeword is selected and introduced in the compressed stream [6]. Figure 2.10 represents the architecture of block-adaptive coder.

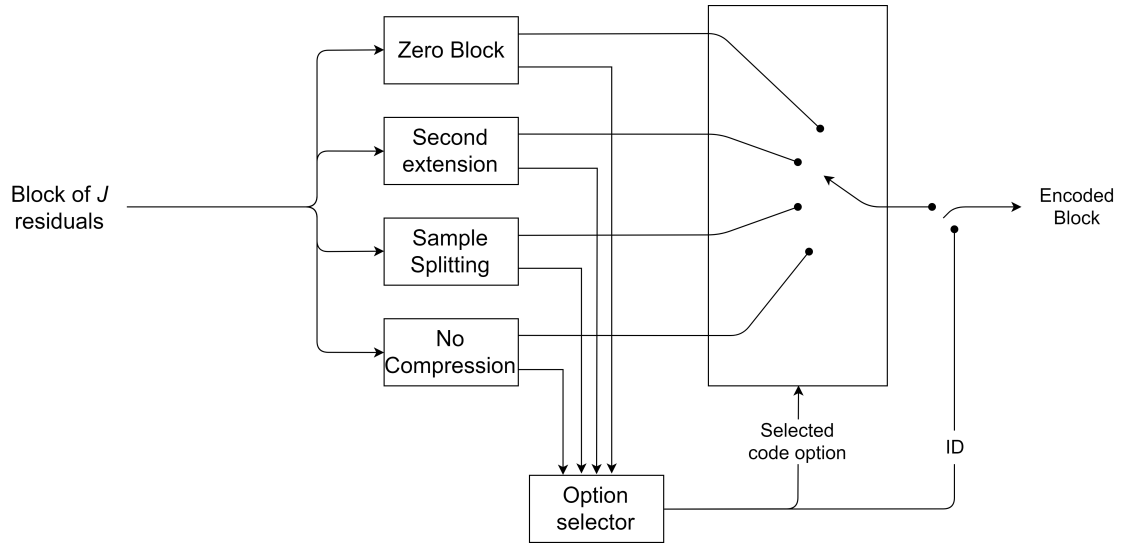


Figure 2.10: Block-adaptive coder diagram, adapted from [6].

The compressed image must be zero padded in the end so that the size is multiple of the output word size B.

There are four algorithms to compress the blocks:

i) Sample Splitting

This option defines a variable k which iterates from 0 to 13. For each iteration, the option selects the k least significant bits (LSBs) from the MPR and encodes it as the binary representation. The remainder value is encoded as the FS codeword as shown in Figure 2.11.

From those 14 iterations, the k value that minimizes the codeword length for the whole block is selected and competes with the others algorithms for the shortest encoded block.

Algorithm 1 shows the pseudo-algorithm for the selection of the shortest codeword.

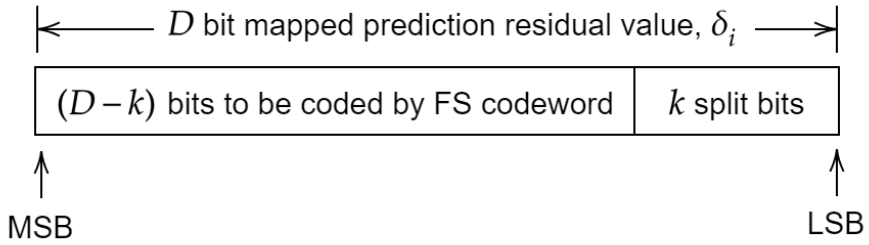


Figure 2.11: Sample splitting codeword for one MPR, adapted from [6].

Algorithm 1 Shortest codeword selection for the sample splitting algorithm

```
for k=0 to 13 do
    total_length = 0;
    foreach residual on block do
        encode_as_binary (residual >> k);
        encode_as_FS (residual & (0x1 << k));
        total_length += encoded_residual (k);
    end
    if (total_length < shortest ) then
        shortest = encoded_residual (k);
    end
end
```

ii) Second-Extension

The second-extension encodes consecutive pairs of MPRs of δ_i and δ_{i+1}:

$$\gamma = (\delta_i + \delta_{i+1})(\delta_i + \delta_{i+1} + 1)/2 + \delta_{i+1}. \tag{2.32}$$

γ is encoded by the FS for every pair of residuals resulting in $J/2$ encoded residuals.

iii) Zero Block

When all residuals on a block have a value of zero, this algorithm encodes the number of consecutive all-zero blocks. This number is converted in FS codewords specified in Table 2.1. The remainder-of-segment (ROS) denotes that the remainder of a segment consists of five or more all-zero blocks.

Table 2.1 Zero-block fundamental sequence codewords as a function of the number of consecutive all-zeros blocks, from [6]

Number of All-Zeros Blocks	FS Codeword
1	1
2	01
3	001
4	0001
ROS	00001
5	000001
6	0000001
7	00000001
8	000000001
.	.
.	.
.	.

iv) No Compression

As the name suggests, the MPR block is not compressed and the values are encoded unchanged.

The zero block algorithm is prioritized over the other functions since it compresses more information than the other options. In the case of the sample splitting and second extension having the same codeword length, the sample splitting is prioritized. Lastly, the block is not compressed, only if the other 3 methods result in a longer codeword.

After selecting the shortest code, the compressed block is inserted in the bitstream following a coded data set (CDS) format. Every CDS starts with an ID field that specifies the method used for the compression. The ID key is represented in the Table C.1 of appendix C.

If the sample splitting option is selected, the coded format for this method is depicted in Figure 2.12. The option ID field contains the value of k, the second field contains all the FS codewords from the block. The second field has the concatenated k least-significant bits from each MPR.

Figure 2.13, shows the data format for the other methods. All methods start with an ID field. When the second extension option is selected, the second field is the FS codewords, as shown in Figure 2.13a. The residuals are encoded in pairs, so only $J/2$ residuals will be encoded per block. If the zero block option is chosen, the field is the FS codewords for the number of all-zero blocks as depicted in Figure 2.13b. Lastly, when no compression is applied, the CDS is composed by the binary representation of J MPRs as in Figure 2.13c

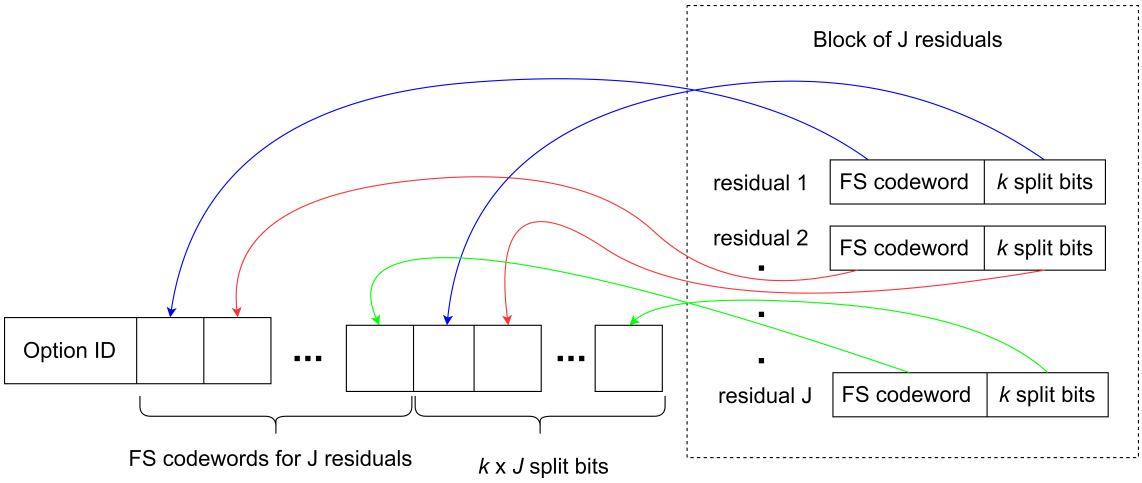


Figure 2.12: Coded data format when sample-splitting option is selected, adapted from [6].

Option ID	FS codes for $J/2$ transformed MPRs

(a) Coded data format when second-extension option is selected, adapted from [6].

Option ID	FS codeword

(b) Coded data format when zero-block option is selected, adapted from [6].

Option ID	J mapped prediction residuals

(c) Coded data format when no compression option is selected, adapted from [6].

Figure 2.13: Coded data format for different methods.

2.3 Summary

This chapter presents an overview of multispectral and hyperspectral imaging and addresses the difference between these two types of images. Two sensors are compared in terms of number of bands and bandwidth, namely the Landsat-8, a multispectral sensor and the hyperspectral sensor AVIRIS.

The CCSDS 123 could be described in two parts. The predictor calculates the residuals, which are the difference between the samples and a prediction of samples. This prediction is calculated from the neighboring samples. Instead of encoding the samples,

this method encodes the residuals, with values close to zero, following a normal distribution instead of a uniform one.

The residuals are encoded in two modes, 1) the sample-adaptive coder, which encodes individual residuals and 2) the block-adaptive coder encodes blocks of residuals using 4 different algorithms, which the smallest resulting codewords are then chosen for the compressed image.

3

Nvidia Jetson TX2 and CUDA

Contents

Graphics processing units (GPUs) were originally designed for rendering graphics and images. Soon the GPUs manufacturers realized the potential for GPUs to solve problems with high volume of information and data independence. Comparatively to central processing units (CPUs), for similar die sizes, GPUs have more arithmetic logic units (ALUs) capable of running 10000 threads concurrently [20], but have higher average memory access times and less memory capacity.

Handheld devices pushed for a new class of GPUs to emerge, focused on bringing high data throughput at low energy consumptions. Nvidia started the development of the Tegra series, a low power system on a chip (SoC) featuring ARM CPUs with Nvidia GPUs in the late 2000's, targeting the mobile devices market. In 2014, Nvidia launched the Jetson series, a low power development board intended to bring GPU compute capabilities to the embedded systems world.

In this chapter is shown an overview of the CPU and GPU architecture followed by the basic principles of compute unified device architecture (CUDA) and its optimization techniques. Lastly, is presented the hardware characteristics of the Nvidia Jetson TX2, the SoC used to parallelize the consultative committee for space data systems (CCSDS) 123 standard.

3.1 CPU Architecture

In current days, most of the computers use a modified version of Harvard architecture, mixing both von Neumann and pure Harvard architectures. The modified Harvard architecture is characterized by having the instruction and data in the same space address but capable of accessing them concurrently. When the CPU loads content from external storage to cache, it behaves like a von Neumann architecture, loading the instructions and data within the same bus. When executing from cache, the CPU acts like a Harvard architecture by accessing different memories for instructions and data.

Instead of requesting content to memory address by address, caches request contents in blocks of addresses, thus taking advantage of spatial locality and temporal locality mechanisms to ensure fast memory access times. Conventionally, CPUs have 2 to 3 cache levels. In multicore systems, each core has 2 different L1 caches, one for instructions and another for data, while L2 or L3 are usually, but not always, accessible by all cores. Figure 3.1 shows a typical CPU cache hierarchy.

Since CPUs use different datapaths for instructions and data, they can achieve instruction level parallelism by using instruction pipelining. The idea behind this concept is to divide instructions into 5 micro-instructions with each occurring simultaneous in the same clock cycle, in an attempt to keep the CPU busy all the time. Superscalar processors im-

plement various pipelines allowing the CPUs to execute various instructions at the same time. Figure 3.2 shows an example of a simple pipeline and a 2-way superscalar pipeline.

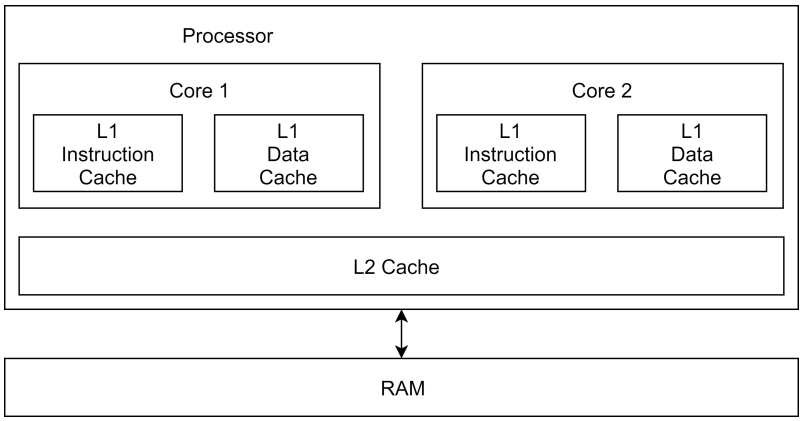
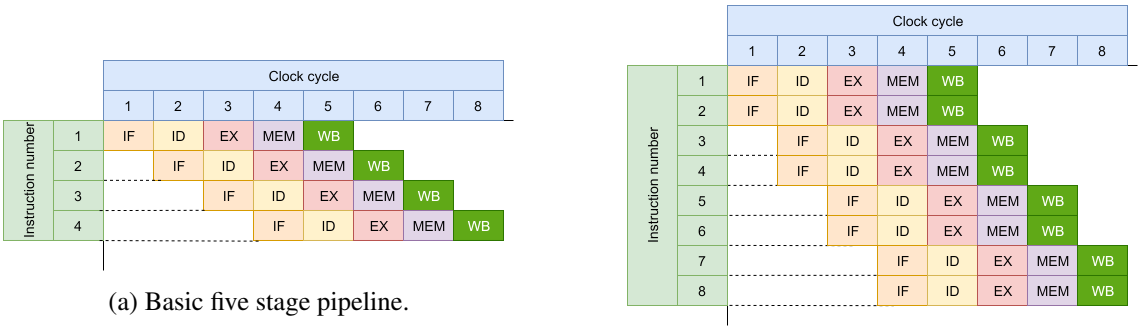


Figure 3.1: Typical multicore CPU cache hierarchy.



(a) Basic five stage pipeline.

(b) 2-way superscalar five stage pipeline.

Figure 3.2: Example of five stage pipelines (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

3.2 GPU Architecture and CUDA

While CPUs are optimized to have low latency, GPU architecture is optimized to have high throughput, allowing to execute various independent operations at the same time. As stated above, for the same die size, GPUs have more computation units compared to CPUs. CUDA cores are the Nvidia's compute units containing ALUs for integer and both double precision and single precision floating points. These CUDA cores are incorporated in streaming multiprocessors (SMs), the basic components of Nvidia GPUs. Each SM contains hundreds of CUDA cores, load/store units, special function units (sin, cos, tan, etc...), thousands of 32-bit registers, shared memory, L1 cache, texture memory, instruction cache, instruction buffers, warp schedulers and two dispatch units [21, 22]. Figure 3.3 depicts an Nvidia's SM architecture, with a CUDA core in detail.

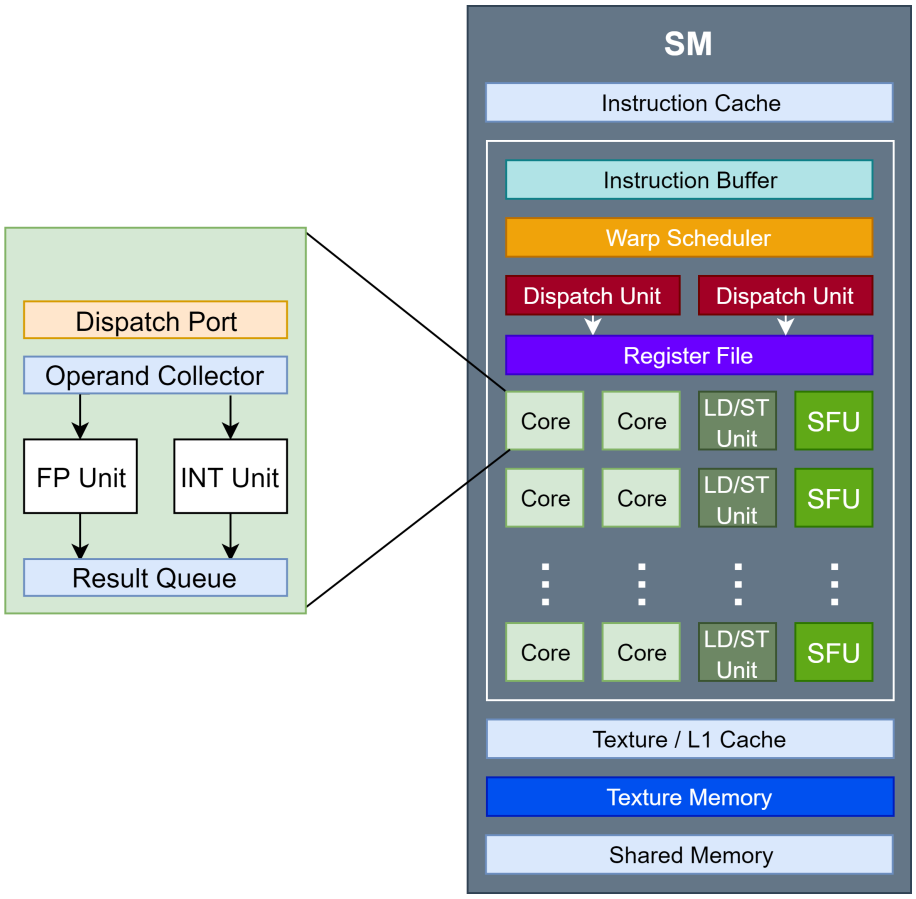


Figure 3.3: Characteristic GPU architecture.

CUDA is an application programming interface (API) developed by Nvidia that enables heterogeneous and parallel programming in Nvidia GPUs using C/C++ and Fortran. This API allows the programmer to write kernels (functions executed on the device (GPU)). The kernel is organized in a grid with blocks executing threads as illustrated in Figure 3.4, the user defines the number of threads for each block and the number of blocks for the grid. Before the host (CPU) launch the kernel, data is usually transferred from the host to the device and transferred back after the kernel finishes. Memory transfers from CPU to GPU have a time overhead associated and in some cases, this overhead may surpass the kernel execution time. When a kernel is launched, each block is assigned to SMs which can execute more than one block. As the blocks terminate execution, new blocks may be launched in vacated SMs [7].

From the hardware perspective, when an SM executes several blocks, it partitions them into warps contains 32 threads. The warps are scheduled by the warp scheduler and executed using a single instruction, multiple thread (SIMT) execution model. If the block dimension is not multiple of 32, some threads in a warp will be idle during the kernel execution. In order to achieve full efficiency, the block dimensions should always

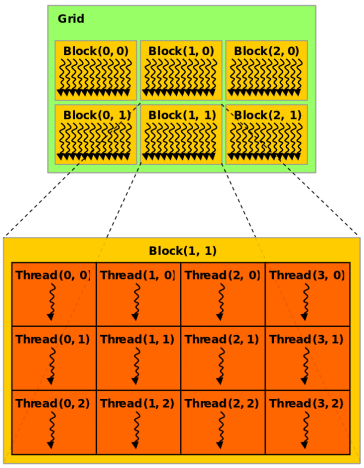
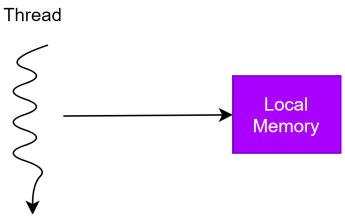


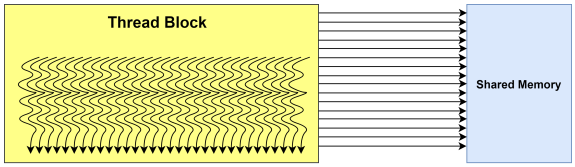
Figure 3.4: Thread execution hierarchy in CUDA, obtained from [7]

be multiple of 32 [7].

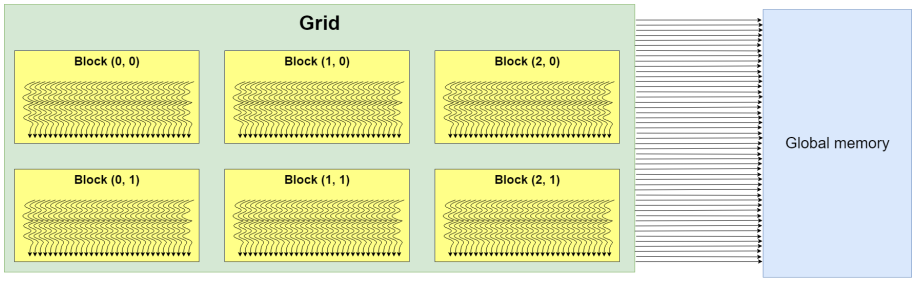
From the memory perspective, each thread has its local memory, usually registers as depicted in Figure 3.5a. This type of memory is the fastest in the GPU, although, it is very limited in terms of space and is private, meaning that other threads can not access it. Figure 3.5b portraits the block-accessible shared memory. Compared to registers is slower and larger, however, it can be accessed by all threads in the same block. Finally, global memory is the slowest and largest GPU memory and is accessible by all blocks and threads as shown in Figure 3.5c.



(a) Thread memory hierarchy.



(b) Block memory hierarchy.



(c) Grid memory hierarchy.

Figure 3.5: CUDA memory hierarchy and visibility, adapted from [7].

3.3 CUDA optimization techniques

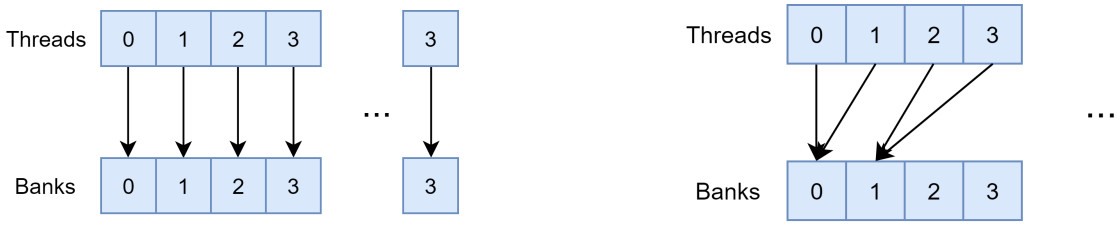
This subsection discusses some optimization techniques and its drawbacks, showing how it can be used to achieve full efficiency on the kernel.

i) Local Memory

The kernel can use the fastest type of memory, the registers. Depending on the kernel launch configuration, the compiler allocates a determined number of registers for each thread. Although, if a thread utilizes all the available registers, it incurs in register spilling and saves data in high-level memory (L1, L2 cache or global memory). Register spilling does not always decrease performance. It may decrease if there is increased pressure on the memory bus or a high instruction count per thread [23].

ii) Shared Memory

Each block can use 48 KB of shared memory for compute capabilities before 7.0 [7]. To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. However, if two addresses are requested in the same bank, as depicted in Figure 3.6b, there is a bank conflict and the access has to be serialized, decreasing throughput. Figure 3.6a represents a memory access without conflicts.



(a) Conflict free shared memory access.

(b) 2-way bank conflict.

Figure 3.6: Types of shared memory accesses.

iii) Constant, Texture and Surface Memory

Constant memory and texture memory is a read-only on-chip memory, that is written before executing the kernel. Constant memory is faster but limited to 64 KB, while, texture memory is larger, optimized for 2D spatial locality and designed for streaming fetches with a constant latency. Texture can achieve better performance if there is locality in the read patterns compared to global or constant memory. Surface memory is the same as texture but can be written and read. Although, it cannot be read and written within the same kernel launch.

iv) Coalesced memory accesses

Memory coalescing refers to combining multiple global memory accesses into a single transaction, meaning that a warp can access 32 sequential words in one transaction, as in Figure 3.7a. The accessed is not coalesced if the memory access is aligned but not sequential, misaligned or if the access is stridden, as described, respectively, in Figures 3.7b, 3.7c and 3.7d.

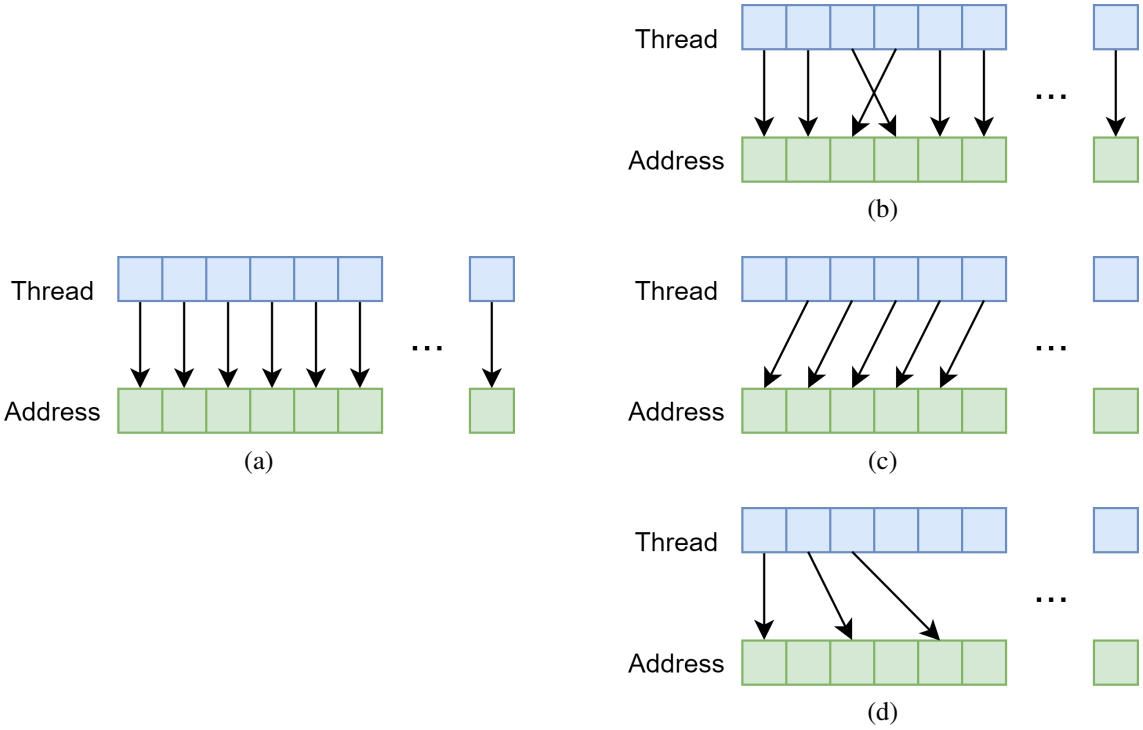


Figure 3.7: a) Coalesced memory access. b) Uncoalesced not sequential memory access. c) Uncoalesced misaligned memory access. d) Uncoalesced stridden memory access.

v) Vectorized memory accesses

This technique uses operations at assembly level to read and write data exploring all the bandwidth from the memory bus. It is different from coalescence, since some coalesced accesses do not consume all the memory bus bandwidth available. This method increases bandwidth and reduces latency but can decrease overall parallelism and increases register pressure [24].

vi) Asynchronous kernel execution

CUDA allows asynchronous memory transfers and execution. The memory transfers and kernel execution is divided by the number of available streams. The main advantage

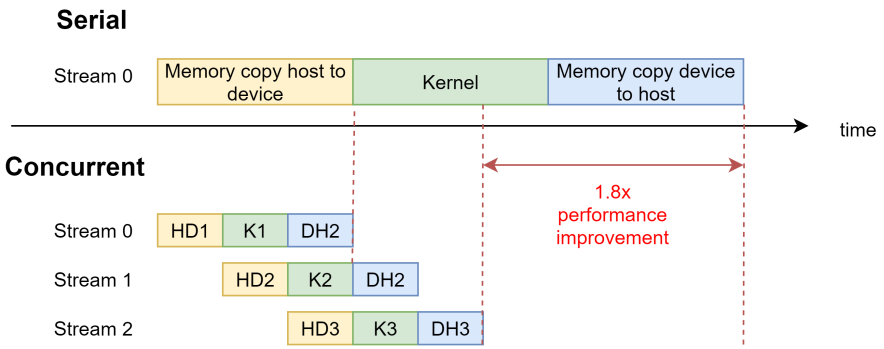


Figure 3.8: Serial kernel execution and concurrent kernel execution with 3 streams overlapped with memory transfers.

of asynchronous launches is that data transfer can be overlapped with kernel execution. Figure 3.8 shows a comparison between serial and asynchronous launches.

vii) Pinned memory

Host (CPU) data allocations are pageable by default. The GPU cannot access data directly from pageable memory. When a data transfer from host to device is invoked, the CUDA driver allocates a pinned array and copy the data to the array, and then transfers it to device memory. The cost of the transfer between pageable and pinned can be avoided by directly allocating arrays in pinned memory. Asynchronous data transfers only work in pinned memory.

viii) Occupancy

Occupancy is defined as the ratio of active warps on an SM to the maximum number of active warps supported by the SM. Occupancy varies over time as warps are launched. Low occupancy may be due to the combination of register usage, shared memory per block and threads per block. If the combination of these factors is not favorable, it may result in poor instruction issue efficiency because there are not enough eligible warps to hide latency between dependent instructions. Fortunately, every CUDA toolkit comes with an occupancy calculator that allows the programmer to parameterize the kernel in order to achieve full occupancy.

ix) Branch Divergence

If threads within a warp diverge via a data depend-conditional branches, the warp executes each branch and disables threads that are not on the path. Distinct warps execute independently regardless of whether they are executing common or disjoint code paths.

The same might occur within blocks. In order for a block to end, all threads in the block must end, and thread divergence may cause some threads to wait for others that took conditional branches.

3.4 Jetson TX2

In order to improve the algorithm's throughput performance, an Nvidia Jetson TX2 is used to implement the CCSDS 123. The board, shown in Figure 3.9, is a development board that runs on a Linux4Tegra (L4T), a linux distribution created by Nvidia.

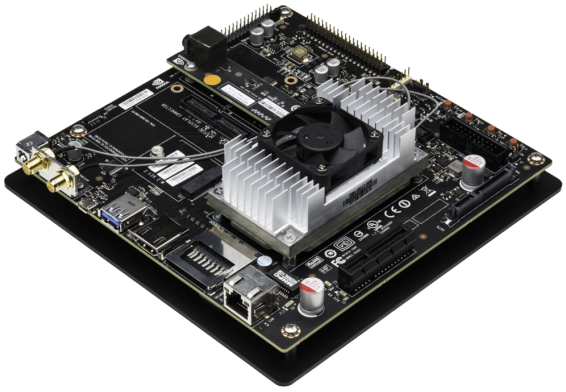


Figure 3.9: Nvidia Jetson TX2 SoC

The system uses an Nvidia GP10B GPU from the Pascal architecture family with 2 SMs containing 128 CUDA cores each. The CPU module is composed by a dual-core 7-way superscalar Nvidia Denver 2 CPU and quad-core ARM Cortex-A57 CPU. The CPUs are connected by a high-performance coherent interconnect fabric developed by Nvidia [8], which allows to share data with reduced overheads. The GPU does not have its memory, instead, this system uses 8GB of low power double data rate 4 (LPDDR4) memory with a 128-bit interface shared between the system. The typical energy usage is around 7.5 watts and is capable of reaching more than one tera floating point operations per second (FLOPS) of performance [25]. The features of this development kit are suitable for the compression of multispectral and hyperspectral images (MHIs) at high throughput performance associated with low-power systems. Table 3.1 shows the specifications of Jetson TX2 and the board overview architecture is represented in appendix D.

Table 3.1 Jetson TX2 specifications, adapted from [8]

GPU	
Architecture	Pascal
CUDA cores	256
Clock	1.3 GHz
SM count	2
Register per Block	32768
L2 Cache	512 KB
L1 Cache	48 KB (per SM)
Shared memory	48 KB (per Block)
Constant Memory	64 KB
Global Memory	8 GB
Memory Bus Width	128 bits
Maximum Bus Frequency	1866 MHz
Copy Engine	1
Storage	
Type	eMMC 5.1 Flash Storage
Bus Width	8-bit
Maximum Bus Frequency	200 MHz
Storage Capacity	32 GB

ARM Cortex -A57	
Instruction Set	64 bit ARMv8.0-A
Cores	4
Clock	2 GHz
L1 cache per core (instruction+data)	48 KB + 32 KB
L2 cache	2 MB
Pipeline	3-Way Superscalar
Nvidia Denver 2	
Instruction Set	64 bit ARMv8.0-A
Cores	2
Clock	2 GHz
L1 cache per core (instruction+data)	128 KB + 64 KB
L2 cache	2 MB
Pipeline	7-Way Superscalar

3.5 Summary

This chapter addresses the hardware characteristics of the Nvidia Jetson TX2 board. It is also shown an overview of the general CPU and GPU architecture and described the superscalar pipeline concept. Moreover, it is presented the CUDA API, its optimizations techniques and described how it operates on the GPU.

4

CCSDS 123 Implementation Overview

Contents

The literature proposes several approaches in order to achieve higher speedups of the consultative committee for space data systems (CCSDS) 123 algorithm. Although, single threaded solutions are already well documented and there is an effort to explore parallel approaches to the algorithm [9], using mainly graphics processing units (GPUs) [1, 9, 15, 26–29] and field-programmable gate arrays (FPGAs) [9, 13, 30–33].

4.1 OpenMP implementations in CPU

An open multi-processing (OpenMP) implementation used in [26], achieved real-time compression and decompression. By ignoring band-parallelism, it achieves better performance than the single core implementation. However, the algorithm scales poorly compared to general purpose graphics processing units (GPGPUs) counterparts. This phenomenon is caused by multi-core systems not being able to achieve the same degree of parallelism as GPGPUs, due to the cost of synchronization between threads and data transfers between the processors, taking more time than GPGPUs.

4.2 FPGA implementations

Space grade FPGAs provide flexible and high performance solutions at low-power with tolerance to space radiation (Virtex 5QV [9, 13, 32]). In [13], it is proposed a low complexity architecture with hardware occupancy between 34% and 44% at a maximum clock frequency of 43 MHz, which was achieved by identifying low complexity parameters that affect the performance of the algorithm. Although low energy consumptions were obtained, compression time increased compared to [34, 35].

Real-time compression was achieved in [9, 31–33, 36] but in [9] older FPGA models suffer from lack of resources, since it cannot deal with very large images. In [32], the authors accomplished high throughput performance without using external memory but incurring in higher energy consumption.

In [28], the authors propose a heterogeneous architecture using an FPGA and a GPU in the same system. The GPU takes advantage of the high throughput performance to compute the calculations, while the FPGA uses the flexible logic for formatting and interfacing the data between the system storage and the GPU.

A low-cost and lightweight solution is proposed in [36] using a lower number of look-up tables, flip-flops and digital signal processor (DSP) achieving a high hardware performance.

4.3 CUDA and OpenCL implementations on GPU

Using an Nvidia GPU in [15], the authors concluded that of all parameters, 3 had the most impact on execution time and compression ability, the number of prediction bands, the prediction neighborhood (column-oriented or neighbor-oriented) and prediction mode (full mode or reduced mode). Reduced prediction mode and column-oriented local sum allows for less computation and increases the compression ratio. As for the number of bands used, it was found that as the number of predicted bands increases, throughput performance decreases, but compression ratio increases. Consequently, there is a trade-off between compression ratio and throughput performance. Two versions were coded in compute unified device architecture (CUDA), one without tiling and another with tiling. In the tiled version, decreasing tile size increases throughput performance but slightly decreases compression ratio.

In [26], by using a GPGPU approach to explore spectral and spatial parallelism of the algorithm, the execution times were superior to the ones developed using OpenMP. The authors exploited the low cost of synchronization between threads, the high data reuse and the high speed of shared memory and cache GPU memory to obtain better performance compared to the OpenMP version.

Writing open computing language (OpenCL) on GPUs is 6 times faster than developing very high speed integrated circuits hardware description language (VHDL) on FPGAs [9]. This paper also showed that FPGAs can run directly connected to the camera sensor, while GPUs need a processor to run in the same way and are susceptible to interruptions or system calls unlike FPGAs. OpenCL is multiplatform supported but is neither faster nor more efficient than using register transfer level (RTL) for FPGAs. However, it is more user-friendly in terms of testing and debugging compared to RTL.

A recent study [29] using a low-power Nvidia Jetson TX1, obtained a higher throughput performance of 124 Msamples/s versus the FPGA implementation at [30], which was 58 Msamples/sec.

4.4 Results from literature

Table 4.1 shows the results for the CCSDS 123 in various platforms from the literature. The CUDA solutions are good candidates for the CCSDS 123 for achieving high throughput performance, but if power consumption is considered, FPGAs usually achieve a superior performance per Watt. The FPGAs are the devices with better energy performance per clock cycle.

Table 4.1 Results for implementations of the CCSDS 123 in various platforms, adapted from [9], N/S stands for not specified. * Higher is better

Platform	Language	Speed (MSa/s)	Power (W)	Efficiency* (MSa/s/W)	Performance per cycle* (MSa/s/W/GHz)
V-5QV FX130T [32]	VHDL	213.00	4.72[3]	45.13	211.86
V-5QV FX130T [13]	VHDL	11.30	2.35[3]	4.81	35.88
V-5QV FX130T [9]	VHDL	179.70	3.04[3]	59.11	N/S
V-4 XC2VFX60 [9]	VHDL	116.00	0.95[3]	122.60	1226.00
RTAX1000S [13]	VHDL	3.50	**0.17**[2]	20.59	502.15
V-4 LX160 [13]	VHDL	11.20	1.49[3]	7.51	56.52
V-5 SX50T [31]	VHDL	40.00	0.70	57.10	1259.93
V-7 XC7VX690T [9]	VHDL	**219.40**	5.30	31.30	N/S
V-4 LX25 [30]	N/S	58.00	1.27	45.67	787.40
Zynq-7020 [33]	VHDL	147.00	0.30	**490**	**3333.33**
GT 440 [9]	OpenCL	62.20	<65.00[1]	0.96	1.19
GT 610 [9]	OpenCL	62.60	<29.00[1]	2.15	2.65
i7-6700 [9]	OpenCL	35.00	<65.00[1]	0.54	0.16
GTX 560M [26]	CUDA	321.91	<75.00[1]	4.29	5.57
2x GTX 560M [26]	CUDA	356.63	<150.00[1]	2.38	3.09
Jetson TK1 [1]	CUDA	3.36	<**2.00**[1]	1.68	1.77
GTX 750ti [15]	CUDA	**401.50**	<60.00[1]	6.69	6.17
Jetson TX1 [29]	CUDA	124.3	<10.00[1]	**12.43**	**12.45**
GTX 580 [27]	CUDA	44.85	<244.00[1]	0.18	0.23
Tesla C2070 [27]	CUDA	30.09	<238.00[1]	0.13	0.11
i7-2760QM [26]	OpenMP	127.89	<45.00[1]	2.84	1.18
2x Xeon X5690 [27]	OpenMP	19.14	<260.00[1]	0.07	0.02

The Xilinx Zynq-7020 [33] is the FPGA with the best efficiency of 490 $MSa/s/W$, compared to the 12.43 $MSa/s/W$ of the low-power GPU Nvidia Jetson TX1 [29] . In terms of throughput performance, the Nvidia GTX 750ti GPU [15] obtained 401.50 MSa/s, while the Xilinx V-7 XC7VX690T [9] reached 219.40 MSa/s.

4.5 Summary

Currently, in terms of throughput per Watt performance, FPGAs dominate the state-of-the-art, but in terms of raw throughput, GPUs are the winners. The advent of low-power GPU potentially increases the efficiency of the algorithm bridging the performance gap between FPGAs and GPUs, putting the latter in a position for competing in terms of performance, cost and development effort.

[1] This denotes the TDP given by the manufacturer
[2] This value was obtained by the Designer SmartPower tool from Microsemi in [13]
[3] This value was obtained by the Xilinx XPower Analyzer in [13]

5

CCSDS 123 Parallelization

Contents

Taking an overall view on the problem, the predictor presents itself as a promising candidate for execution on graphics processing unit (GPU). This is justified by data independence between samples. A particular case arises when the prediction model only uses values from the same band ($P = 0$). This case can remarkably improve throughput performance. The encoder lacks the data independence in the bitstream formation due to the encoder generating variable-length codewords, although some parts can be parallelized.

5.1 Predictor Parallelization

Before creating the parallel predictor, data dependencies must be assessed. Besides the critical path data dependencies, there is one data dependency loop in the weights calculation. From (2.25) for a determined value of t, the current weights vector and local differences vector are used to calculate the next weights vector in each band, meaning that the weights calculation can only be parallelized by band. Nevertheless, the other kernels are independent and can be parallelized by sample. Figure 5.1 proposes a parallel algorithm using 3 kernels: a pre-weights calculation kernel, a weights kernel that encapsulates the data dependency loop, and another post-weights kernel outputting the mapped prediction residuals (MPRs). A fourth kernel is represented in the figure, which corresponds to (2.24) used in the weights calculation. This block does not have any dependencies and can be launched with 1024 threads per block, ensuring maximum occupancy. The result is a static array in the t domain (2.1) with size $N_x \times N_y$.

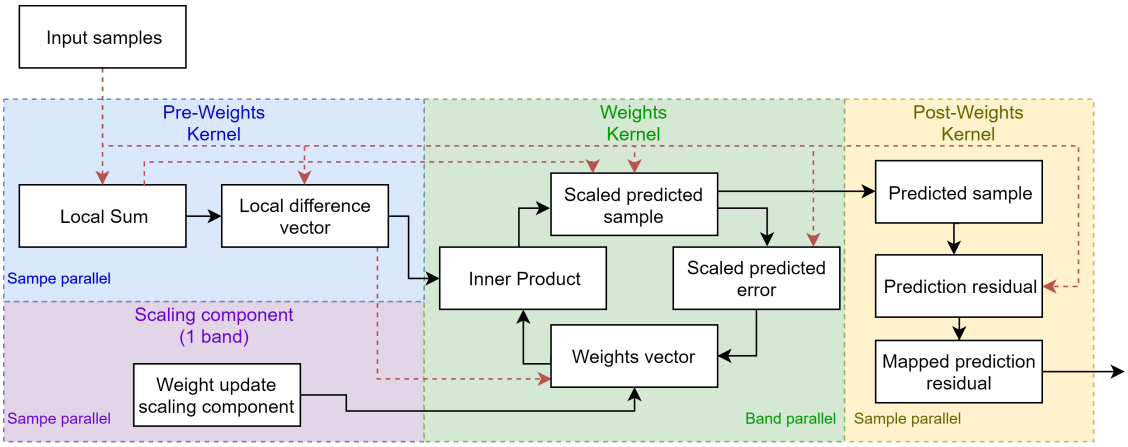


Figure 5.1: Proposed CCSDS 123 parallel predictor diagram. Black arrows represent the critical datapath, while dashed red arrows are indirect data dependencies.

The original samples are used in 3 kernels and, since the image is static, it might be beneficial to use faster read-only memory instead of global memory. Using constant memory is not possible since multispectral and hyperspectral images (MHIs) would not fit. Another option is to use texture memory.

In order to utilize all the memory bus width when reading data from global/texture memory, vectorized accesses are utilized by packing data in such a way that the packed samples match the bus width size. On the Jetson TX2, the bus size is 128 bits or 16 bytes. The dynamic range of the CCSDS 123 is from 2 to 16 bits and, fortunately, almost all sensors that employ this algorithm use 16 bits (2 bytes) for sample representation. In C language, it is possible to manipulate bits but is very inefficient. A better solution is to use predefined types such as chars (1 byte) or shorts (2 bytes). With this perspective, packing 8 samples forces the GPU to use all of the bus width available in a single memory transaction.

Another question to solve is in which order the samples shall be packed. The packing should be done so that each thread processes a packed sample. Looking to weights kernel, which is band-parallel, packing samples in z domain would not bring any benefits since, at least, 8 parallel threads would read the same data. Since the kernel works on the t domain, packing samples in order to y would require extra memory and increases divergence. The best option is to group in order to x as shown in Figure 5.2. The pre-weights and post-weights kernels do not have data dependencies and can be parallelized in other orders, but since the weights kernel is restricted to the x axis, the others kernel must be grouped in the same axis. Even though those packed samples can be converted between kernels, the performance does not increase because of the overhead to execute that operation.

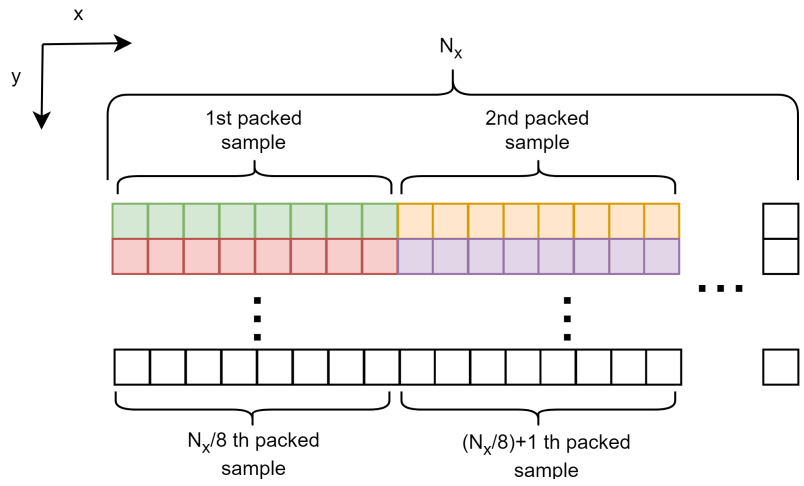


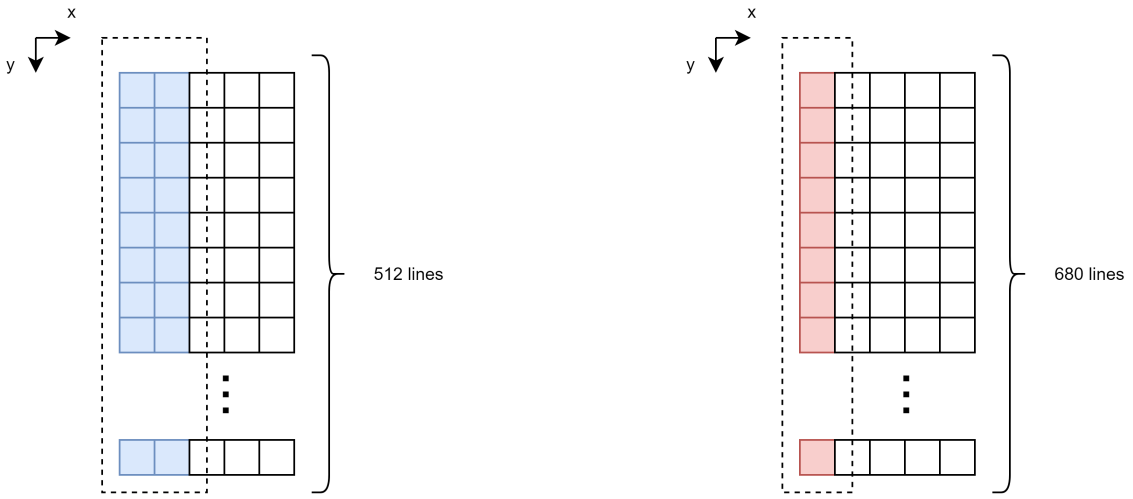
Figure 5.2: Diagram of grouping samples with 16 bits to groups with 128 bits

The first problem to emerge is how to read data from the GPU. In some kernels, storing data in local memory does not introduce any advantages since it increases the number of global memory accesses. In such cases, a better strategy is to load data to shared memory.

For the pre-weights kernel, each thread computes a packed sample. Depending on which mode the predictor is running (column-oriented or neighborhood-oriented), each

thread reads from global memory 2 to 5 times. Reading that data to local memory would consume too many registers and might induce register spilling. In order to reduce global memory reads, each thread loads the sample to shared memory accommodating whole columns in the same band, reducing branch divergence.

Ideally, each column should have a number of elements that is a power of two, so each block has a number of threads that is also a power of two. Figure 5.3 shows an example of blocks with different number of threads. The local sum and local difference vector use values from the sample itself and its neighbourhood (Figure 2.5).



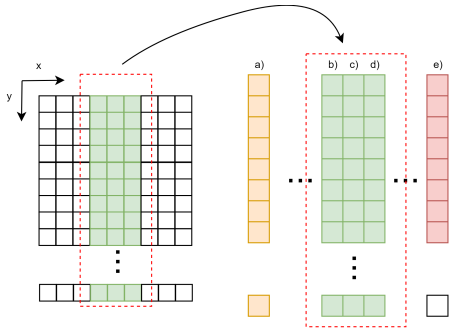
(a) Block processing 2 columns with 1024 samples

(b) Block processing 1 column with 680 samples.

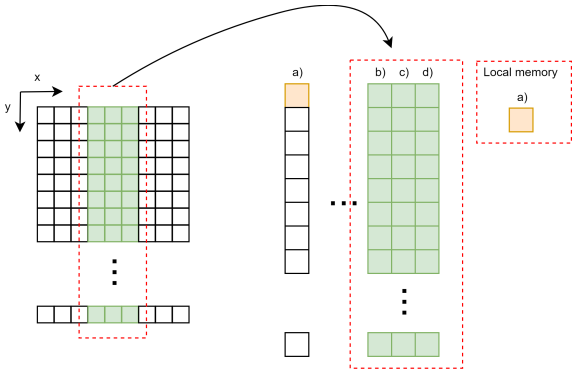
Figure 5.3: Number of columns processed in a block. Each block can execute at most 1024 samples

When the predictor runs on the neighbourhood-oriented mode with shared memory, each thread makes one global memory access and reads the neighbor samples from shared memory as depicted in figure 5.4a, except when sample is in the border of the block, then, the thread makes one more global memory read. If the block executes one column, each thread reads 3 times from global memory. In column-oriented mode, as illustrated in Figure 5.4b, the border threads do not need to load to do extra global memory reads. From (2.3), the particular case in this is in the first line, which loads the $x-1$ element. This value is read to a register, even though it increases branch divergence, reduces shared memory consumption. The local sum and local differences results are stored in registers and written to global memory using vectorized writes.

In the weights kernel, since there is a data dependency loop, the kernel must be band-parallel. The kernel is launched with one block with one thread per band as shown in Figure 5.5. Since each band is parallelly executed, every band has a weight vector with P elements which use registers and is initialized by (2.12).



(a) Block processing 3 columns in neighbourhood-oriented mode. One thread executes a green sample. Threads in border columns make to 2 global memory reads while inside thread make 1 global read.



(b) Block processing 3 columns in column-oriented mode. Green samples are loaded to shared memory by its correspondent thread. The yellow sample in column a) is loaded to a register by the first thread in column b).

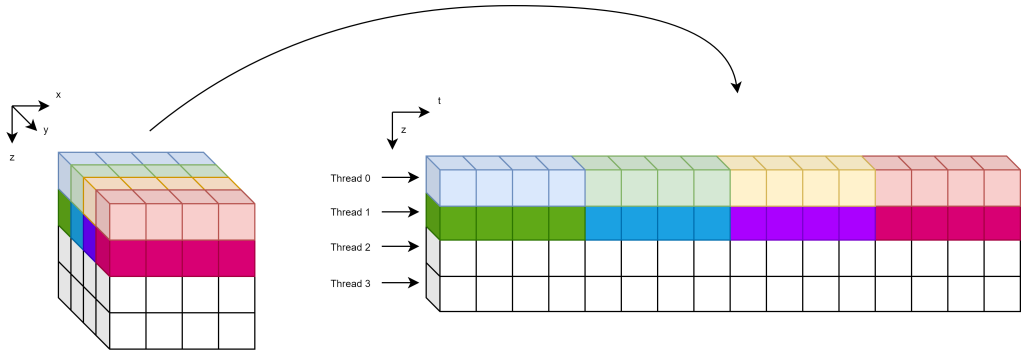


Figure 5.5: XYZ domain transformation to Zt domain for the weights calculation kernel.

The local difference vector reads values calculated by the pre-weights kernel, more specifically, it reads P values from the previous bands. In this context, for reducing the number of global reads, the local differences are loaded to shared memory. Unfortunately, using this method incurs in shared memory bank conflicts. Surprisingly, it is better than having repeated global memory reads. One solution would be to have P arrays of shared memory to store the same local differences, however, for high values of P and images with high number of bands, the kernel would consume a large amount shared memory. Other variables calculated in this kernel use registers.

The occupancy of this kernel relies heavily on the number of bands of the image. Despite the fact that the number of registers per thread can be controlled to achieve better occupancy, in most cases is not enough to achieve full occupancy.

The post-weights kernel is easily parallelized. Each thread executes one scaled predicted sample from the weights kernel. All variables use registers and branch divergence is present but it is inevitable.

5.1.1 Particular Case ($P = 0$)

When the predictor runs on the reduced mode and does not use any bands for prediction ($P = 0$), the kernel can be simplified. In (2.11), the vector is constituted by the values from previous bands. When no previous bands are used ($P = 0$) the vector is empty. This condition implies that the result of (2.14) does not exist and the local differences vector is not needed. Figure 5.6 shows predictor block diagram under reduced mode for $P = 0$.

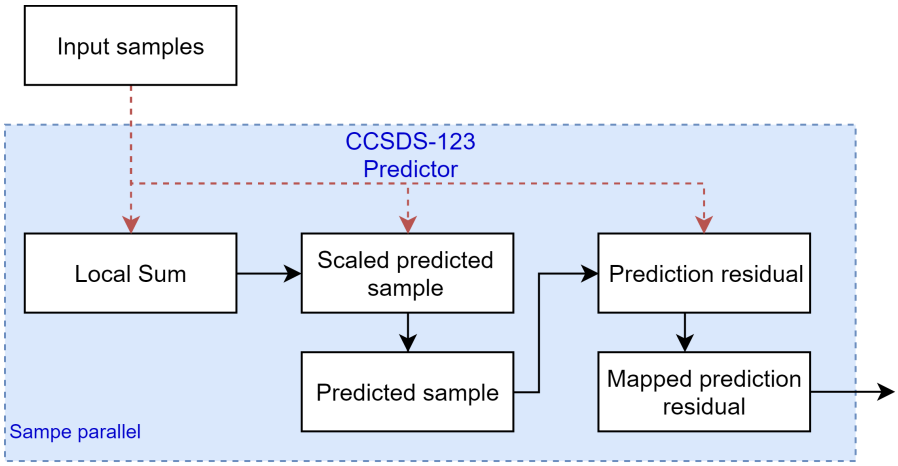


Figure 5.6: CCSDS 123 predictor when no bands are used in the prediction.

As a result of eliminating the data dependency loop, the predictor can be executed in one kernel. This reduces the overall overheads of calling the kernel compared to the previous version. In (2.15), the second member is eliminated since it never verifies the condition $P > 0$. This elimination is essential, removing the need for values from different bands.. As a consequence, the samples can be packed in order to z for preventing data dependencies in the local sum calculation. Figure 5.7 illustrates a diagram with the packing of the samples in the z axis.

This configuration does not make the best prediction since it does not use inter-band prediction [14] ($P > 0$) but can be promising in terms of throughput performance due to data independence.

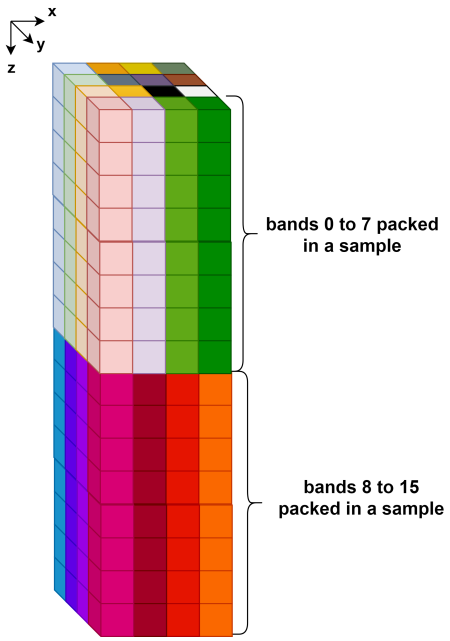


Figure 5.7: Diagram of packing samples with 16 bits to 128 bit words across bands in the z axis.

5.2 Encoder Parallelization

Since the length of codewords varies, the encoder is hardly parallelizable. However, it is possible to extract some parallelization from the bitstream generation. The rule of thumb is to distribute the bitstream generation between the compute units and, in the end, concatenate the various bitstream in one. This part may require bit-wise operations since the C language does not allow for direct operation at bit level. Figure 5.8 shows an example of a concatenation of 2 unaligned bitstreams.

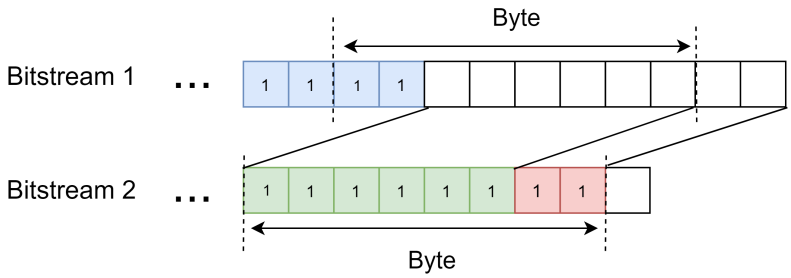


Figure 5.8: Bitstream concatenation. The green bits of bitstream 2 must be shifted 2 bits to the left and added to the byte in the bitstream 1. The red bits from bitstream 2 are filtered with an AND mask, shifted to the left 6 bits and added to the second byte in bitstream 1.

Jetson TX2 has 2 different central processing units (CPUs) with different characteristics. The Denver 2 CPU is 7-way superscalar with 2 cores and the ARM CPU is 3-way

superscalar with 4 cores. For some types of codes, such as loops or batch operations, superscalar CPUs can issue multiple instructions on data in the same amount of time. Although, if mechanisms of branch prediction fail, the pipelines must be emptied and this can be very penalizing. Compared to the Denver 2 CPU, in the 3-way ARM pipeline, branch prediction misses are not so penalizing, but it cannot execute so many instructions at the same time.

When parallelizing the encoder, the loads in the cores must be balanced. Depending on the code and the type of CPU, the cores can process more or fewer instructions. The main goal is that every core should take the same time to process a variable amount of data and achieve the best throughput performance. This leads to the problem of balancing the loads between cores. There is no exact way to automatically balance the cores, unless if there is previous knowledge about the image. The load needs to be manually tuned to find the best ratio. Another way to automatically tune the loads would be to formulate an equation in predictor.

The sample adaptive encoder encodes each band separately in the t domain, achieving band level parallelism. To parallelize this encoder, 2 approaches are made: 1) using six cores to encode various bands and merge the resulting bitstreams into one, as shown in Figure 5.9. The biggest drawback of this method is the difficulty of balancing the loads in each core. 2) using the GPU by launching a number of threads equal to the number of bands as depicted in Figure 5.10. Each thread executes one band and generates the bitstream. In the end, the CPU concatenates all the bitstreams together. This method is easily parallelized but can potentially turn slower than CPU, since the GPU clock and transfer time imply penalties.

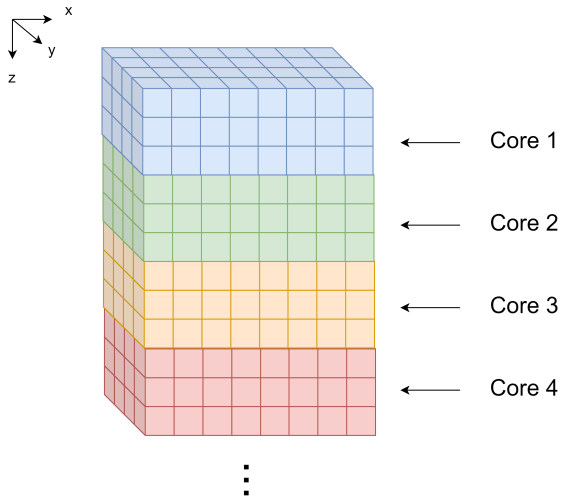


Figure 5.9: Sample adaptive encoder parallelized in the CPU. Each core encodes a determined number of bands.

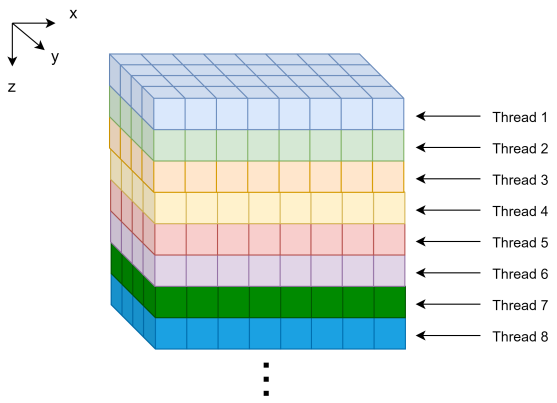


Figure 5.10: Sample adaptive encoder parallelized in the GPU. The kernel is launched with a number of threads equal to the number of bands.

In the block adaptive encoder, different algorithms are applied to the blocks of MPRs as depicted in Figure 2.10. Each of those algorithms is assigned to a compute unit in order to ensure the lowest processing time. After each method outputting the option selected for each block, those are divided between cores to generate the bitstream. As in the sample adaptive encoder, when all the cores finish processing, the bitstreams are merged. Figure 5.11 shows a basic scheme for the parallel block adaptive encoder.

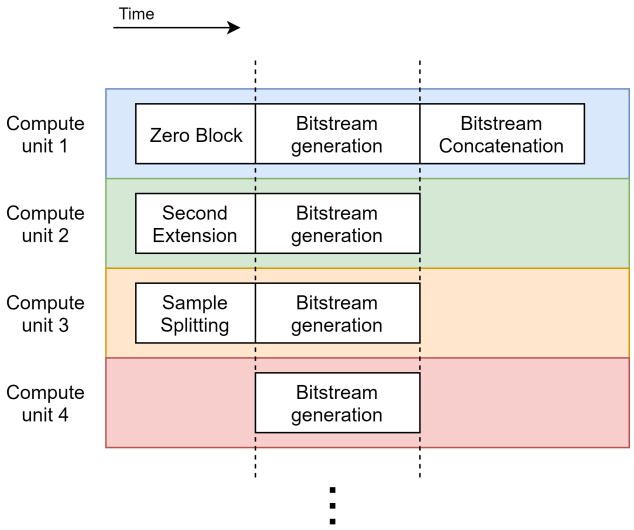


Figure 5.11: Parallel block adaptive encoder .

5.3 Summary

This chapter explains the parallelization on the CCSDS 123 for the predictor and the 2 types of encoders. It is also discussed a particular type of case when the predictor only uses samples from the same band ($P = 0$), simplifying the predictor.

6

Experimental Results

Contents

This chapter presents the achieved results from the parallelization of the CCSDS-123. Each experiment is executed 20 times to ensure the result consistency. The standard deviation is calculated but is negligible and, therefore, not represented in the figures. All obtained results can be found on [37].

6.1 System Setup

The Jetson TX2 has temperature sensors to trigger the fan when temperatures rise. The temperature values are registered in files by operative system (OS) [38]. In order to ensure consistency, each run is executed at the same temperature, 39ºC for the central processing unit (CPU) and 38ºC for the graphics processing unit (GPU). The fan velocity is controlled using pulse width modulation (PWM) which the operating system reads from a file [38]. To control the temperature, a bash script that checks the temperature and writes the PWM value for controlling the fan velocity.

In this thesis is used an implementation of CCSDS 123 made by european space agency (ESA), downloaded from [39]. This code implements a serial version. All the speedups and the parallelizations are calculated from this code.

The serial version has macro NO_COMPUTE_LOCAL. When this macro is deactivated, the local differences are pre-computed and stored in a local buffer but consume large amounts of memory. Since the system has a limited amount of memory, the macro is activated in all tests and the local differences are computed whenever needed [40].

The serial version is compiled by gcc for the native architecture (-march=native) and with level 2 optimizations (-O2). The tests are executed using the command *taskset –cpu-list*, which executes the in a defined core, and the command *nice -n* which gives the program maximum priority.

The execution times were measured using *clock_getime* from time.h C library. All the images tested in this thesis were downloaded from [41] and the dimensions and sizes can be found in appendix E.

The tests for the predictor were done on reduced mode and column-oriented local sums. The other parameters used are $R = 32$, $\Omega = 4$, $t_{inv} = 2048$, $v_{min} = -6$ and $v_{max} = -6$. These parameters chosen due requiring less computation effort.

The parameters for the encoders, use the same parameters for predictor with $P = 0$. The block adaptive encoder uses $B = 1$ and $J = 64$. The sample adaptive encoder runs with $U_{max} = 8$, $\gamma^* = 9$, $\gamma_0 = 8$, $k_z^{'} = 14$.

The serial times were attained by executing the functions in one Denver 2 CPU core. For counting the number of instructions, it was used *nvprof* in the GPU and the *perf* command for the CPU.

6.2 Predictor

In section 5.1.1, intra-prediction ($P = 0$), has the best potential for parallelization. The approach taken to parallelize the predictor is to analyze how each optimization technique improves the execution time.

Firstly, the basic functions from the predictor are profiled to see which have the larger execution time and to decide which ones will be initially parallelized. Figure 6.1 shows the execution times for a total of 13.01 seconds. The first 4 bars, from left to right, correspond to (2.3), (2.15), (2.20) and (2.21) respectively. The remaining bars represent (2.22) with the scaled predicted value sign and the delta absolute value corresponding $(-1)^{\widetilde{s}_z(t)}$ and $|\Delta_z(t)|$ respectively.

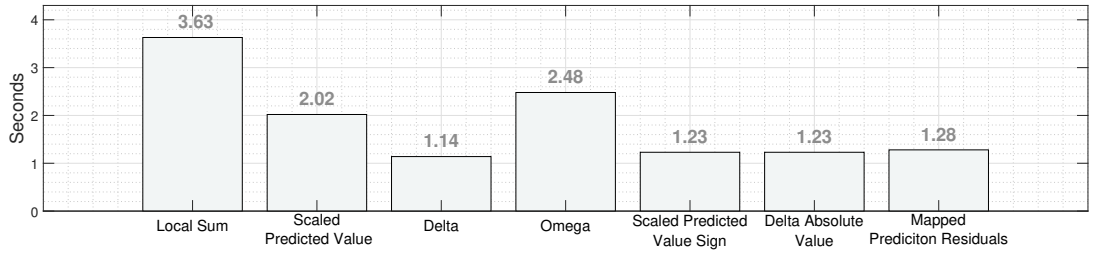


Figure 6.1: Function profiling times in the serial predictor for the AVIRIS Hawaii image ($P = 0$).

Before starting exploring the proposed optimizations, it is analyzed how the kernel launch parameters and types of memory impacts the execution time. In Figure 6.2 is depicted the execution times for different threads per block and the use of pinned memory. This kernel does not use any optimizations such as shared memory and vectorized accesses. Using the maximum number of threads per block with pinned memory improves the execution time by 7.73 times compared to the serial version.

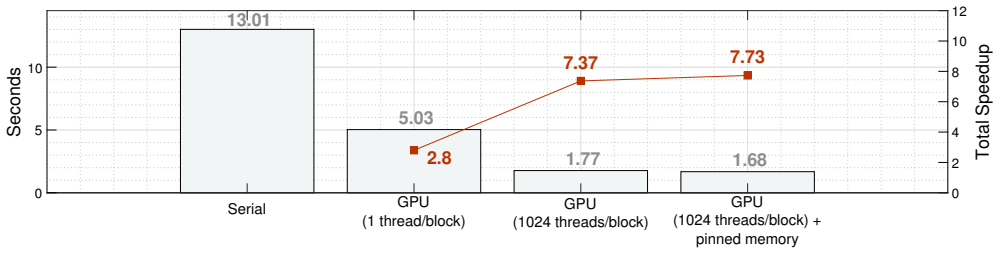


Figure 6.2: Predictor execution times for AVIRIS Hawaii, with GPU improvements. The red squares represent the speedup in relation to the serial time.

Figure 6.3 shows the times for transferring the same amount of data from host to device (HtoD) and device to host (DtoH). Using pinned memory does not affect the transfer time from HtoD but greatly affects time between DtoH.

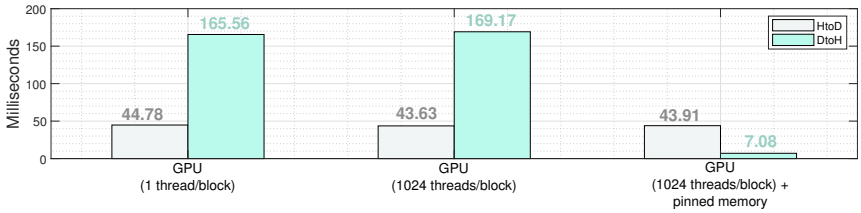


Figure 6.3: HtoD and DtoH transfer times for the same amount of data.

In CUDA, 2 types of texture memory can be used. Texture reference is the traditional way of using textures. Texture objects are a new type of memory that does not require manual binding and unbinding, and behave like C++ objects being handled as if they were pointers [42]. Figure 6.4, using both types of memory, yields the same execution time, although, texture objects scale better than texture references [42], and for that reason, they will be used in the rest of the experimental results. Surface memory is not used since it would be hard to implement and cannot be read and written in the same kernel since the cache is not kept coherent [7], thus, eliminating memory reuse.

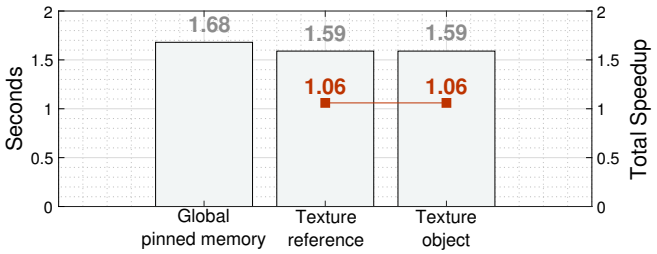


Figure 6.4: Texture reference and object against global pinned memory. The red squares represent the speedup in relation to the global pinned memory.

The parallelization is produced using global pinned memory and texture objects. From this point on, the proposed optimizations will be progressively applied. Figure 6.5 shows the execution times when shared memory is utilized. The optimization is applied to store the image, and then, consecutively until the mapped prediction residuals (MPRs) calculation. The squares represent the speedups to their respective base versions without optimizations. The texture memory version, typically, achieves faster execution times. This happens due to texture memory residing inside the device (faster reading times) and by exploiting the spatial locality compared to global memory [7].

The second optimization employed is vectorized accesses. One version processes 4 samples per thread, while another processes 8 samples. The 8 samples version uses all the available memory bus width, which increases register pressure and shared memory utilization. Processing 4 samples might be beneficial since it can provide more flexibility.

Vectorized accesses might lead to an inefficient kernel, by causing register spilling

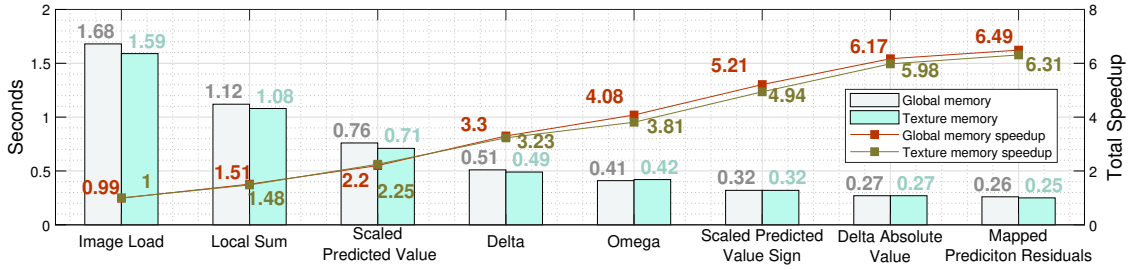


Figure 6.5: Shared memory applied to global memory and texture memory. The optimization is applied cumulatively from left to right. The speedups lines are in relation to the basic version on global memory and texture object from Figure 6.4.

or preventing the kernel from executing because it uses significant amounts of shared memory per block. In order to avoid these problems, the shared memory will only be used to store essential variables, as long as, the requested memory per block does not surpass the maximum allowed value (48 KB).

To avoid register spilling, the variables can be stored in shared memory but this can decrease the occupancy by half. When blocks run on 48 KB of shared memory, the kernel cannot achieve full occupancy. To achieve full occupancy, each block should have at most 32 KB of shared memory. Another option to avoid the drop in occupancy is to split the kernel and use a maximum of 32 KB of shared memory.

In the following figures, to assess this phenomenon, 2 types of tests were made: using kernels with 48 KB of shared memory and using kernels with 32 KB of shared memory. For processing 4 samples per thread, Figure 6.6 shows the results when using global memory and Figure 6.7 when using texture memory.

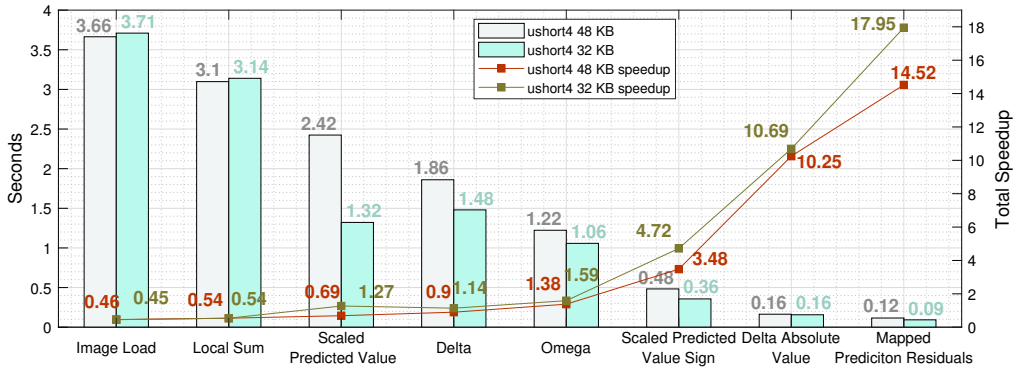


Figure 6.6: Execution times when the kernel processes 4 packed samples stored in global memory. The speedups lines are in relation to the basic version on global memory and texture object from Figure 6.4.

In both cases, using split kernels (32 KB), marginally improves the speedups. Using texture memory is slightly better than global memory. In scaled predicted value calcula-

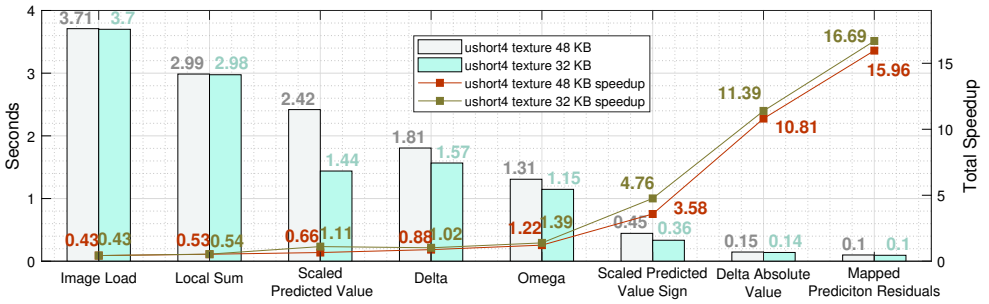


Figure 6.7: Execution times when the kernel processes 4 packed samples stored in texture memory. The speedups lines are in relation to the basic version on global memory and texture object from Figure 6.4.

tion, there is a notable discrepancy between the 48 KB version and the 32 KB version. The only difference between the kernels is that the scaled predicted values are stored in shared memory in 48 KB version while the 32 KB version used registers.

Figures 6.8 and 6.9 show the times and speedups for threads executing 8 samples. Although the first functions take more time to execute compared to the 4 samples version, the MPRs function breaks the record by achieving a time of 87.9 ms for the global memory version and 85.96 ms for the texture memory version. The speedups for global and texture versions are higher than 4 samples version. Also, as expected, the 32 KB version is faster than 48 KB version.

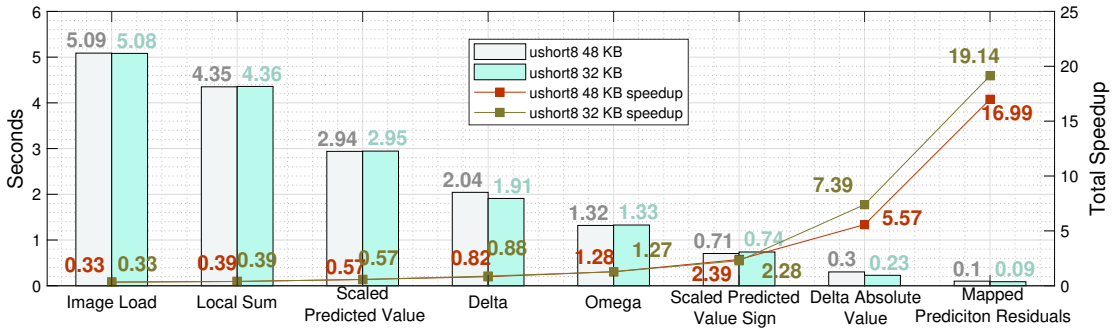


Figure 6.8: Execution times when the kernel processes 8 packed samples stored in global memory The speedups lines are in relation to the basic version on global memory and texture object from Figure 6.4

Next, is applied asynchronous launches and data transfers. This method is applied to the best executing times from the 8 and 4 packed samples versions. Each stream process a number of bands. In Figure 6.10, in the 8 samples version, using global memory, improves by 24.8 times concerning to the time obtained in Figure 6.4. In Figure 6.11, the execution time on the device decreases almost by one quarter from the 4 samples version to the 8 samples. The DtoH transfer time also decreases due to the kernel using the data bus

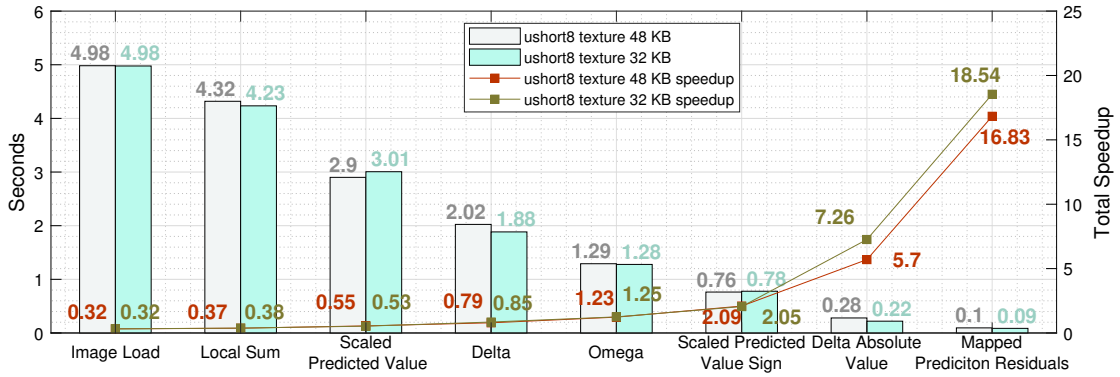


Figure 6.9: Execution times when the kernel processes 8 packed samples stored in texture memory. The speedups lines are in relation to the basic version on global memory and texture object from Figure 6.4

more efficiently. For an unknown reason, the overhead in the 8 samples texture version increases. This might be due to synchronization and memory management and is also observed in the version with 14 and 7 streams.

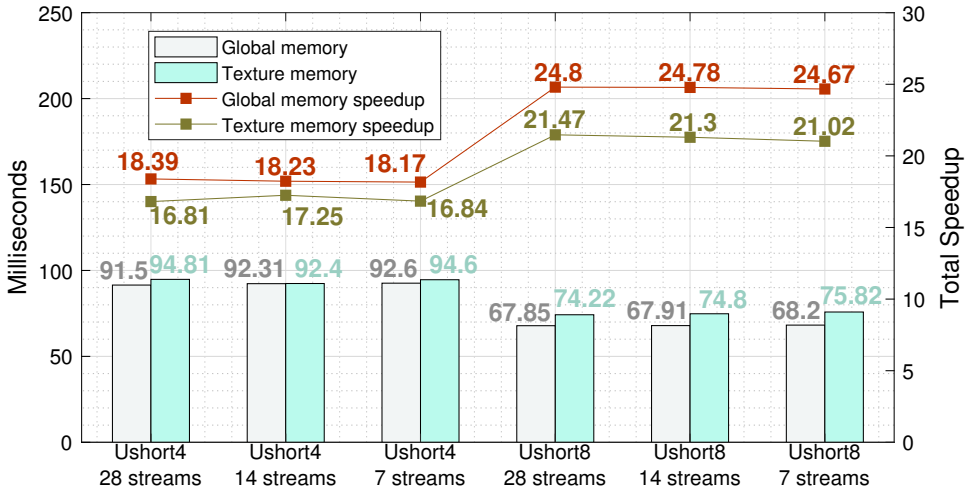


Figure 6.10: Execution times for asynchronous kernels with 28, 14 and 7 streams. The speedups lines are in relation to the basic version on global memory and texture object from Figure 6.4

Figure 6.12 shows how the different types of optimizations influence the data transfer times. From Figure 6.3 it was concluded that the pinned memory affects the HtoD time. The vectorized accesses improves DtoH times. Usually, HtoD transfers using texture memory are slower than global memory, this happens due to textures being bound to CUDA arrays. These arrays are indexed in 1D, 2D or 3D and access times are slightly slower than linear memory [7]. Using streams, the times increase slightly due to overhead and synchronizations between streams.

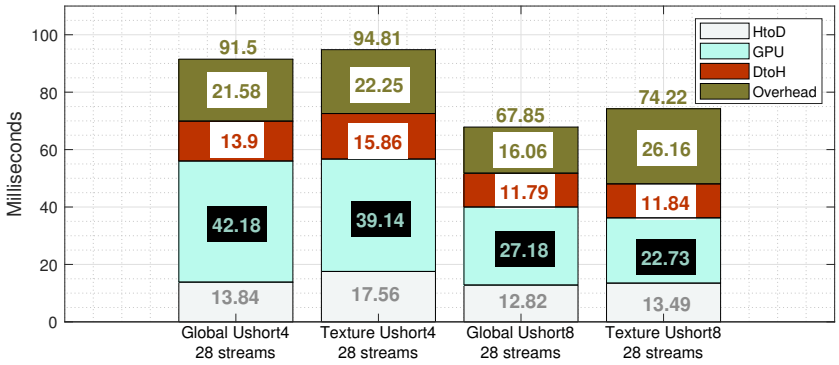


Figure 6.11: Asynchronous kernel launch and transfer times from Figure 6.10 in detail.

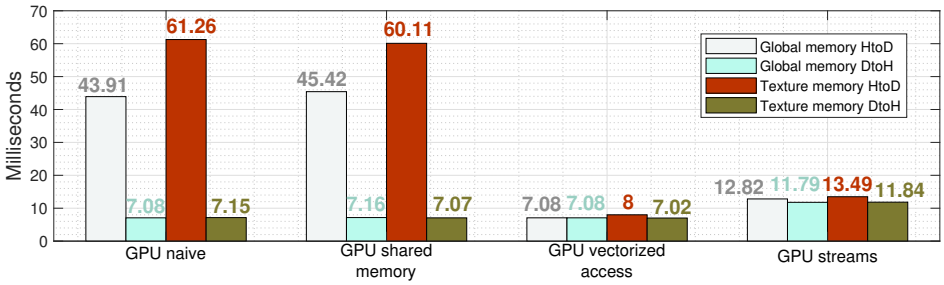


Figure 6.12: Data transfer times between optimizations.

Finally, in Figure 6.13 it is represented the overall speedup to the serial version. Using 28 streams in global memory and processing 8 samples per thread, it is achieved a speedup of 191.81 times. Analyzing the speedup lines, texture memory achieves a better result for all optimizations except when using streams.

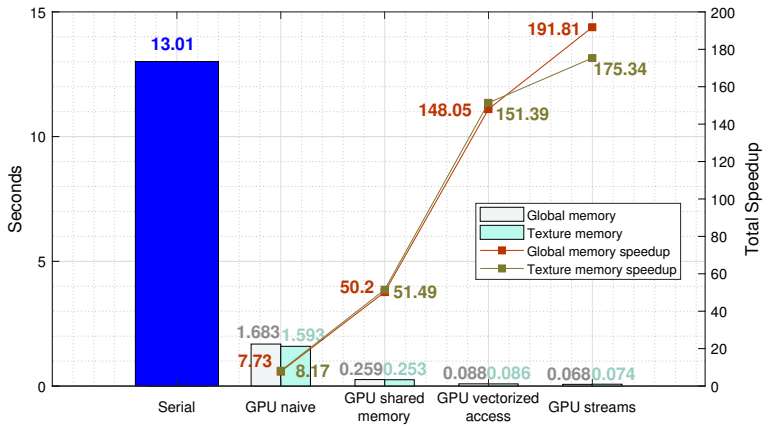


Figure 6.13: Overall speedups in the CCSDS 123 predictor implemented in the GPU.The speedups lines are in relation to the serial version.

In Figure 6.14 it is shown the roofline model for the GPU optimizations. The serial

point was executed in the CPU, and, compared to the GPU points, this point has more operations due to the loops in the algorithm. The measure also may be skewed as a consequence of being measured with a different command. The kernel increases the performance by reducing memory transactions, following the philosophy of single instruction, multiple data (SIMD), reducing intensity.

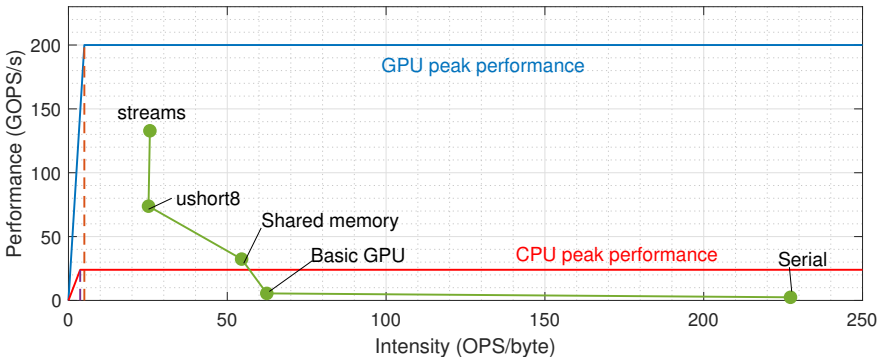


Figure 6.14: Roofline model for the optimizations applied to the predictor.

These tests were conducted in the AVIRIS Hawaii image. For other images, Figure 6.15 shows the obtained results. The best speedups are from the AVIRIS images. The remaining images obtained gains around 150 times. This is due to problem mentioned in Figure 5.3. The image dimensions do not allow to launch the kernel with a maximum number of threads. Nevertheless, the gains are significant. The multispectral images were not tested due to not having more than 8 bands. Tests were done for the texture memory. However, the attained results were not better than its counterpart. The roofline model for the various images is described in Figure 6.16, showing a good performance that is limited by thread divergence.

All kernels developed in this thesis follow the procedure described above. The parallelization for the inter-band predictor ($P > 0$) from Figure 5.1 is also analyzed.

In Figures 6.17 and 6.18 illustrate the serial execution times in various images. As expected, the times follows a linear increase as P rises, except for the 3 largest images in Figure 6.17. For $P > 4$ in the AVIRIS Yellowstone and for $P > 14$ in the compact reconnaissance imaging spectrometer for mars (CRISM) sensor images, the times stop following a linear increase. This may be explained by cache misses that happen especially in large images. The characteristics of the image also influence the prediction execution time.

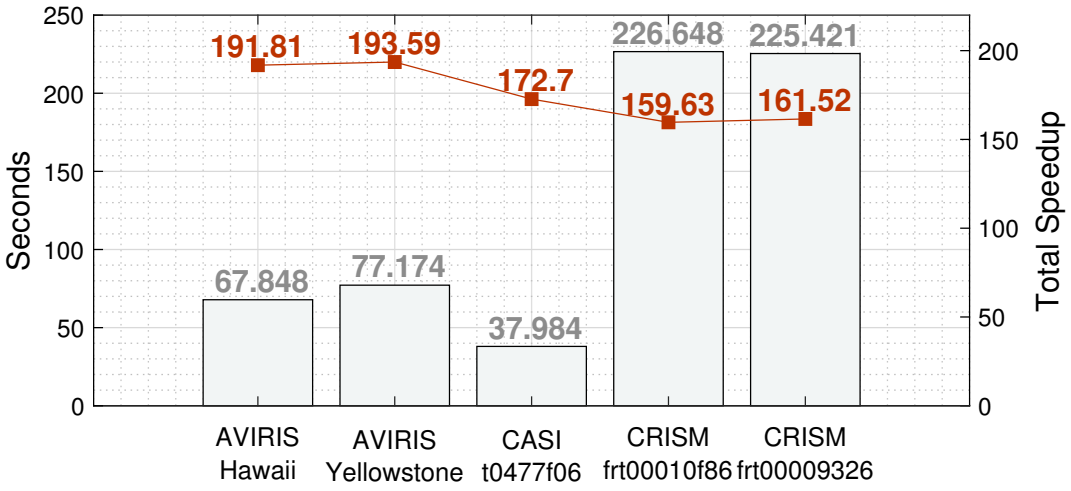


Figure 6.15: CCSDS 123 predictor implemented in the GPU with $P = 0$ for various images. The speedup line is compared to the serial version.

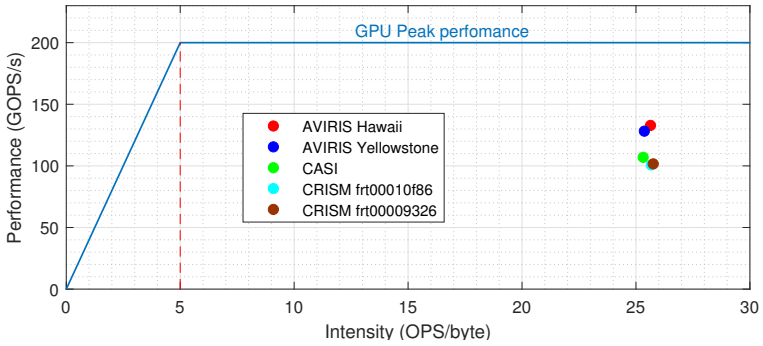


Figure 6.16: Roofline model for the parallelized version of the predictor.

Figures 6.19 and 6.20 represent the execution times for the parallelized predictor running on the GPU. The time evolution follow a linear pattern but the cache miss effect takes place when $P > 6$ for the images of the CRISM sensor. In multispectral images, the times do not follow a linear increase. This can be due to cache misses but also explained by shared memory bank conflicts. These problems are present in the hyperspectral but since multispectral have less bands, this effect is more pronounced.

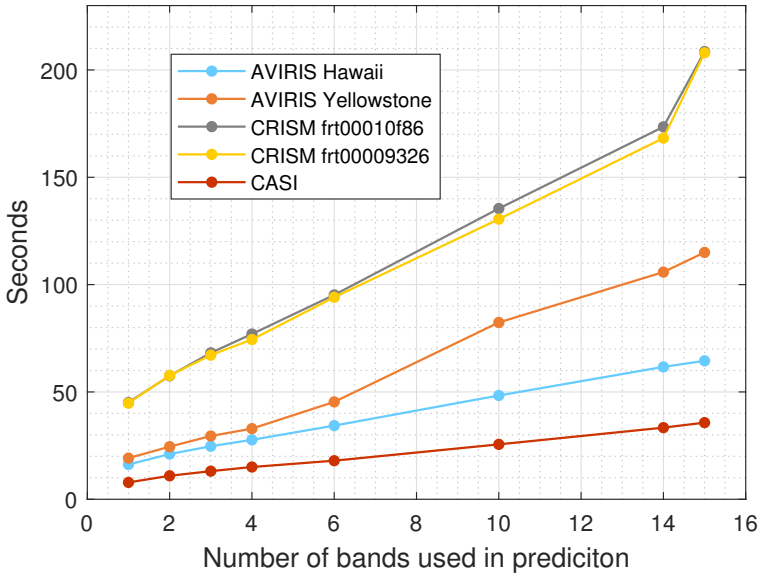


Figure 6.17: Execution times for the serial CCSDS 123 predictor for $P > 0$ in hyperspectral images.

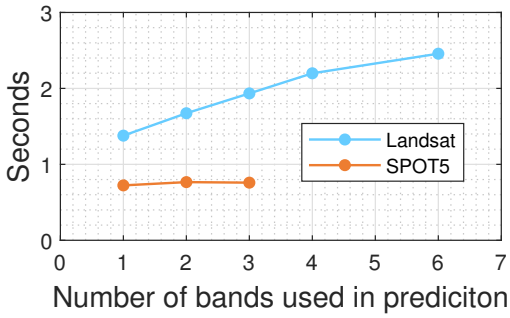


Figure 6.18: Execution times for the serial CCSDS 123 predictor for $P > 0$ in multispectral images.

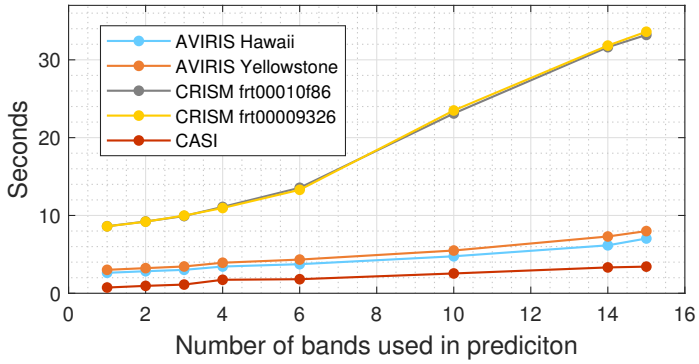


Figure 6.19: GPU execution times for theCCSDS 123 predictor for $P > 0$ in hyperspectral images.

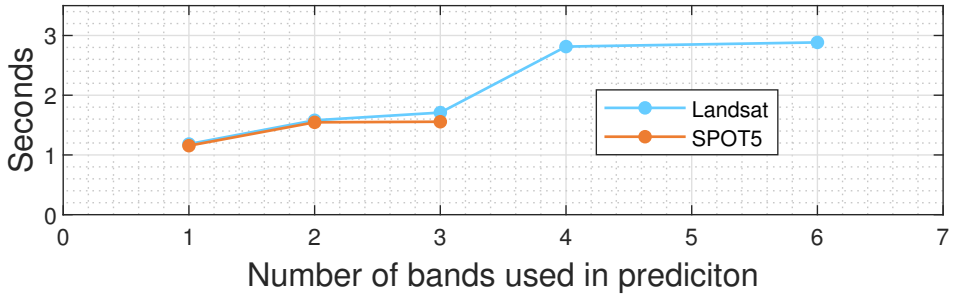


Figure 6.20: GPU execution times for the CCSDS 123 predictor for $P > 0$ in multispectral images.

The speedups for this kernel are calculated in Figure 6.21. The first thing to notice is that the multispectral images did not obtained the desired performance. Many of those runs resulted in speedups below the horizontal red which represents a unitary speedup. However, the hyperspectral images obtained speedups between 5 and 15 times.

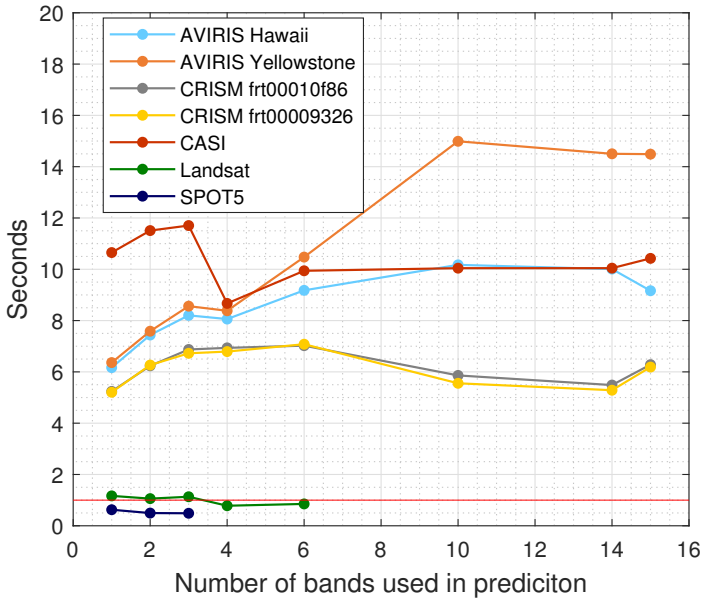


Figure 6.21: Speedups for the CCSDS 123 predictor. The red horizontal line stands for the speedup equal to one. The Landsat image has 6 bands while the Toulouse image has 3.

Upon analyzing Table 6.1, some of the kernels parallelized did not achieved full occupancy. The occupancy on the pre-weights kernel is influenced by the number of rows in the image, as explained by Figures 5.3 and 5.4b. The number of threads running on the weights kernel must equal to number of bands in the image. The worse occupancies are from the CRISM images and multispectral images. The multispectral image have an inherently poor occupancy due to the kernel executing less than 10 threads.The CRISM

images, despite having 545 threads per block, have a low occupancy due to how threads, shared memory per block and registers per thread interferes in division of the streaming multiprocessor (SM), as shown in Figure 6.22.

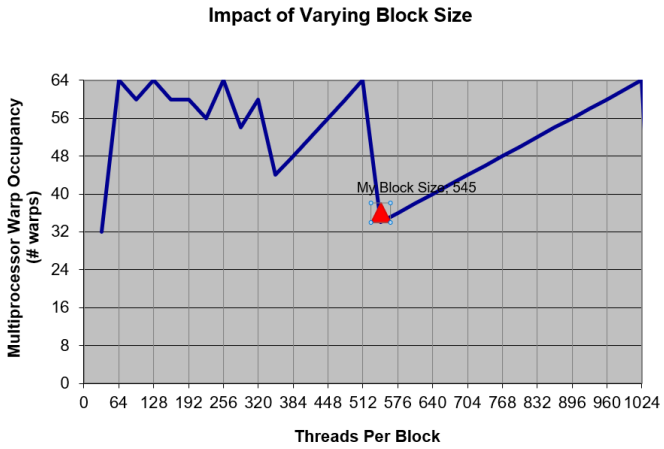


Figure 6.22: Influence on the occupancy for the weights kernel in the number of threads per block vaires for images from the CRISM sensor. The red triangle shows the actual occupancy. Image taken from the CUDA occupancy calculator from the CUDA toolkit 9.0

From Table 6.1, the weights kernel consumes between 89.5% and 99.56% of the predictor time. This means that this kernel is very sensitive to its occupancy. In fact, from Figure 6.21, for $P < 3$, the images with the highest occupancies also have higher speedups. For $P > 3$, other events take control of the speedups, such as caches misses and shared memory bank conflicts. The speedups stagnate for $P > 3$ with three exceptions. The first is the speedup increase in images from the CRISM sensor. This happens because of the drop in performance on the serial version. The second is the compact airborne spectrographic imager (CASI) image, which the speedup drops for $P > 4$, this is explained by inconsistencies in time of the GPU version. The last is AVIRIS Yellowstone, which the speedup increases until $P = 10$ because of the instability in the serial version. However, in [14], for most images, using more than 3 bands for the prediction, does not bring any significant differences in the compressed bit rate performance.

[4]Values are from 1 band to 6 bands
[5]Values are from 1 band to 3 bands

Table 6.1 Table for the occupancy rates and times in percentage for the kernels represented in Figure 5.1. The occupancy of the kernels that are omitted are 100%. The times of the scaling component kernel are very close to 0% and therefore, not represented. The occupancy rates were measured using the CUDA occupancy calculator from the CUDA toolkit 9.0

Image	Pre-Weights kernel occupancy	Weights kernel occupancy	Pre-Weights kernel time (1 band - 16 bands)	Weights kernel time (1 band - 16 bands)	Post-Weights kernel time (1 band - 16 bands)
aviris hawaii	*63%*	*88%*	*4.32% - 1.54%*	*94.66% - 98.1%*	*1.02% - 0.36%*
aviris yellowstone	*69%*	*88%*	*4.01% - 1.47%*	*94.99% - 98.15%*	*1.00% - 0.37%*
crism frt00010f86_07	*100%*	*56%*	*3.46% - 0.83%*	*95.78% - 99.00%*	*0.77% - 0.18%*
crism frt00009326_07	*100%*	*56%*	*3.46% - 0.82%*	*95.75% - 99.00%*	*0.79% - 0.18%*
casi t0477f06-raw	*81%*	*94%*	*8.14% - 1.58%*	*89.5% - 97.91%*	*2.35% - 0.51%*
Landsat agriculture	*100%*	*50%*	*0.85% - 0.36%* [4]	*98.96% - 99.56%* [4]	0.17% - 0.07% [4]
toulouse spot5	*100%*	*50%*	*0.44% - 0.34%* [5]	*99.45% - 99.57%* [5]	0.09% - 0.08% [5]

6.3 Block Adaptive Encoder

The parallelization of the block adaptive encoder follows the architecture from Figure 5.11. The encoder is parallelized by distributing the load between the compute units using portable operating system interface (POSIX) threads.

Figure 6.23 shows a diagram for the solution used in the encoder. The first phase applies a pre-processing to determine the shortest codeword for each block using the 3 methods. From the 3 algorithms executed concurrently, the most expensive is sample splitting. Fortunately, this algorithm is easily parallelizable and runs on the GPU with one thread per MPR block, achieving faster times compared to running in CPU. For these methods, the Denver 2 CPU achieves better performances related to the ARM CPU and, for that reason, the remaining methods are executed, each one, on a Denver 2 core.

Then, in the second phase, the bitstream generation is divided between the cores using threads. Each core has to access the information calculated in the previous phase, in order to identify which method produced the shortest codeword. Due the Denver 2 being 7-way superscalar CPU, the MPR blocks to be encoded are divided roughly by 30% for each Denver 2 CPU core and 10% for each ARM CPU core.

After the bitstream generation, one Denver 2 core concatenates the bitstreams from the previous phase into a single one using bitwise shifting, as explained in Figure 5.8. The CASI image does not qualify for this encoder since the total number of samples is not divisible by 64.

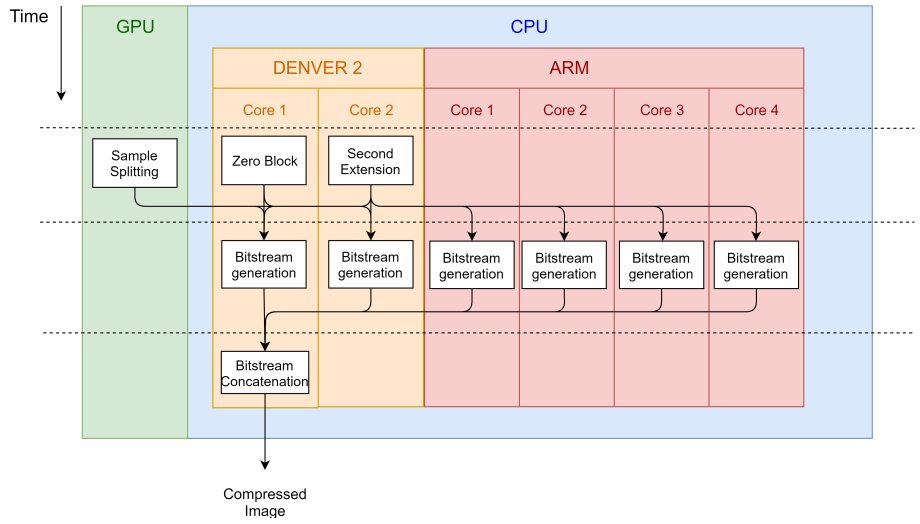


Figure 6.23: Diagram of the parallelized CCSDS 123 block adaptive entropy coder.

Figure 6.24 shows the times for the encoder before the parallelization. Figures 6.25 and 6.26 shows the resulting times and speedups for the parallel version for the hyperspectral and multispectral images respectively. The chart is divided into the 3 phases: the sample splitting, second extension and zero block compose the first phase, ARM and Denver 2 the second and the third is the bitstream concatenation. The second and third phases only start when all functions from the previous phases are finished. Figure 6.25 shows good results for the balance between the CPU cores with the difference being approximately 100 ms. In Figure 6.26 a good balance is harder to achieve between the CPUs cores. Nevertheless, speedups of 9.05 and 7.54 were obtained.

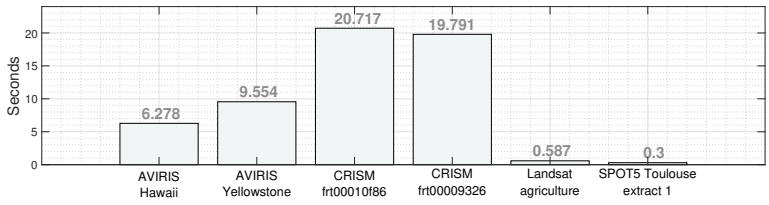


Figure 6.24: Execution times for the serial version of CCSDS 123 block adaptive encoder.

The parallelized block adaptive encoder achieves a great performance shown by Figure 6.27. The points surpass the CPU peak performance due the sample splitting block being executed in GPU. This function represents between 30% to 40% of the execution time, which is significant amount. Executing this block in the GPU shows how powerful the GPUs are when executing operations per second.

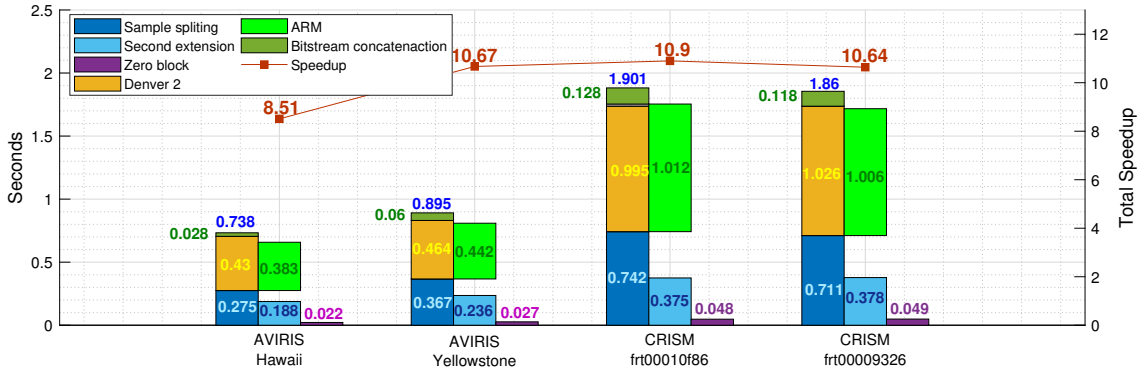


Figure 6.25: Execution times for in detail for the CCSDS 123 block adaptive encoder for hyperspectral images. The chart is divided into 3 phases and one only starts after all functions from the previous phases had finished.

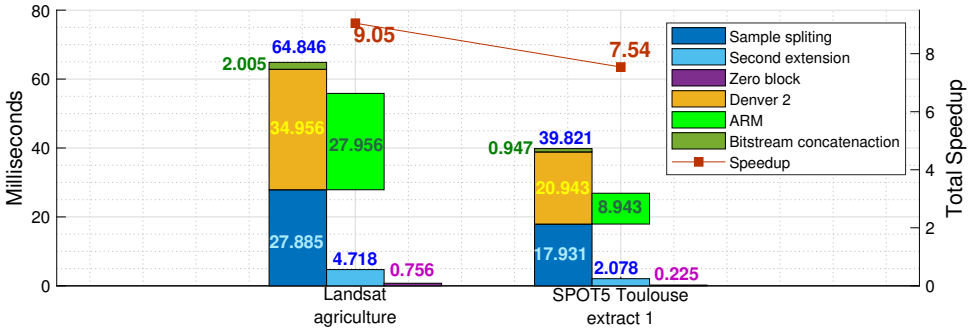


Figure 6.26: Execution times for in detail for the CCSDS 123 block adaptive encoder for multispectral images. The chart is divided into 3 phases and one only starts after all functions from the previous phases had finished.

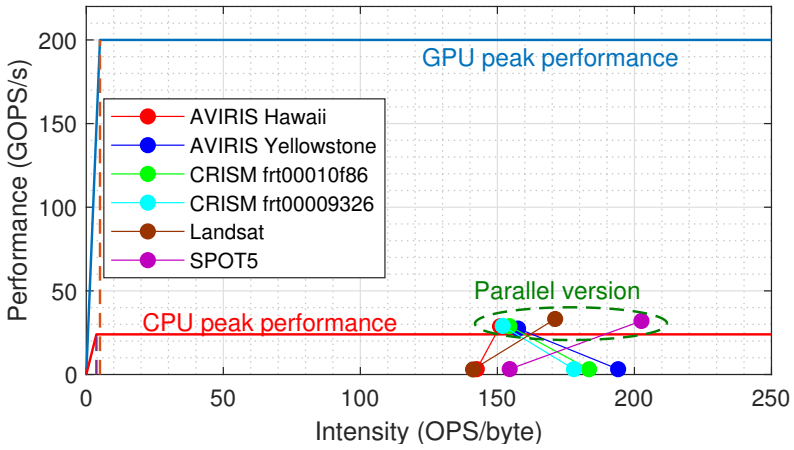


Figure 6.27: Roofline model for the parallelized version of CCSDS 123 block adaptive encoder.

6.4 Sample Adaptive Encoder

The parallelization for this encoder follows the approaches from Figures 5.10, using one thread per band, and 5.9 using one core to process various bands. Figure 6.28 illustrates the serial times executed in one core.

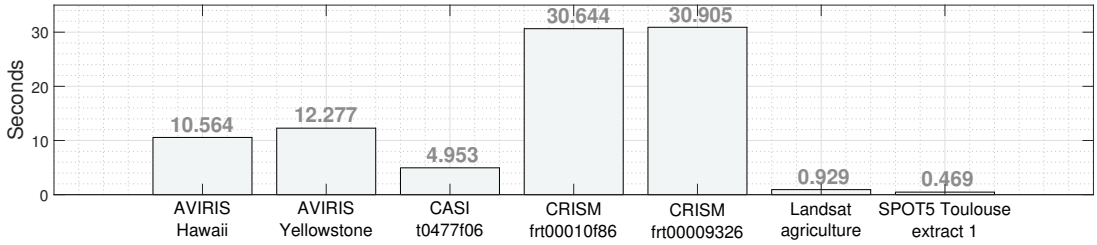


Figure 6.28: Execution times for the serial version of CCSDS 123 sample adaptive encoder.

In figure 6.29 is represented the execution times obtained for the GPU version. The kernel is similar to the weights kernel in the predictor, launching one thread per band. The results for the AVIRIS images are satisfactory, achieving speedups of approximately 3. For the CASI image, it was obtained a speedup of 1.76. This unexpected, since, from the table 6.1, the kernel's occupancy is 94% and the speedup should be similar to those the AVIRIS images. The characteristics of the image itself can explain this phenomenon. For the CRISM images, very low speedups were obtained due to the low occupancy and by the images requiring too much memory to the point of consuming all available random access memory (RAM) memory, causing stalls to the OS. The multispectral images did not obtained good speedups as a result of the kernel running with only 6 and 3 threads.

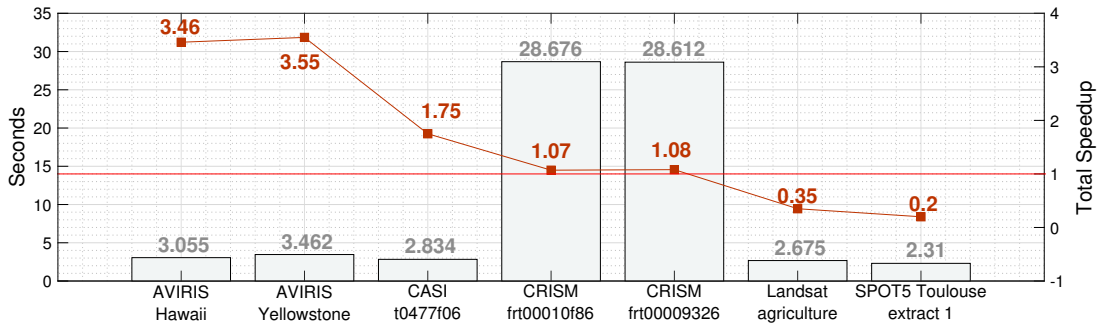


Figure 6.29: Speedups and execution times for the GPU version of the CCSDS 123 sample adaptive encoder.

In Figures 6.30 and 6.31 is depicted the results for the parallel CPU version. As in the previous section, the chart is divided into 2 phases: the ARM and Denver 2 in one, and the

second phase composed of the bitstream concatenation. The results obtained show better results than the GPU version.

In Figure 6.30 all speedups are higher than 5, except for the CASI image. Firstly, the loads in this image are harder to balance between the CPUs cores due to the low number of bands. This type of image takes more computation effort to be encoded by this algorithm. In Figure 6.31 it is not possible to balance the loads. The Landsat image only has 6 bands, which means that each band is processed by one core. The SPOT5 image only has 3 bands, meaning that the first phase, equals the processing time in one of the ARM CPU cores. Nevertheless, the speedups are better than the GPU version. From this conclusion, the weights kernel in the predictor could be executed in CPU instead of the GPU. The roofline model for this optimization is described in Figure 6.32.

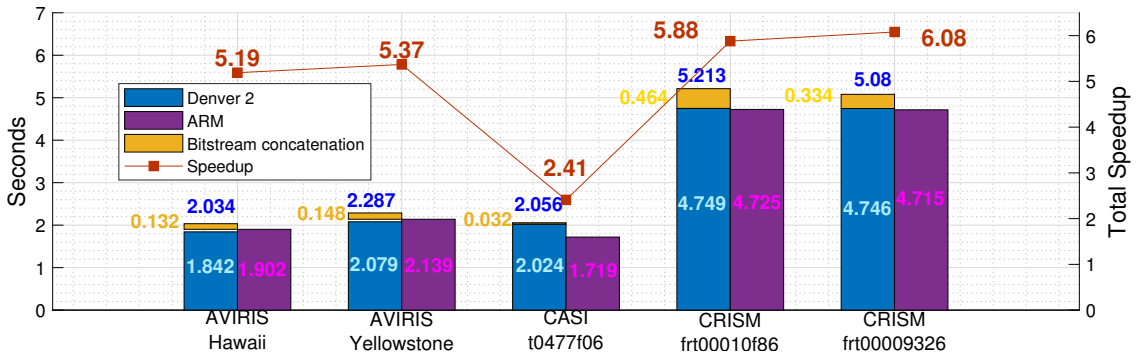


Figure 6.30: Execution times in detail for the CCSDS 123 sample adaptive encoder running on the CPU for hyperspectral images. The bands executed in each core are: AVIRIS Hawaii - Denver 2=48, ARM=32; AVIRIS Yellowstone - Denver 2=48, ARM=32; CASI - Denver 2=4, ARM=16; CRISM frt00010f86 - Denver 2=120, ARM=76; CRISM frt00009326 - Denver 2=120, ARM=76;

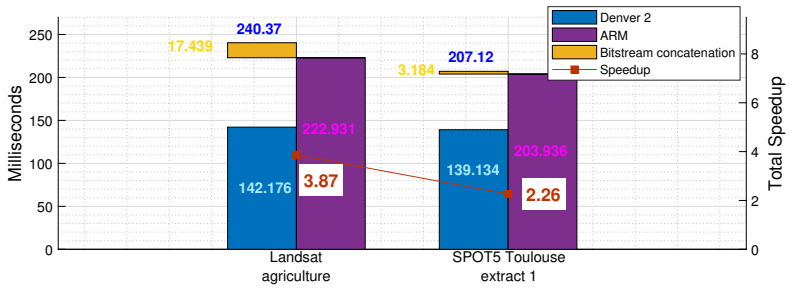


Figure 6.31: Execution times in detail for the CCSDS 123 sample adaptive encoder running on the CPU for multispectral images. In the Landsat image, each core processes a band while in the SPOT5 image, 2 bands are executed in the Denver 2 CPU and the last one executed in one core of the ARM CPU.

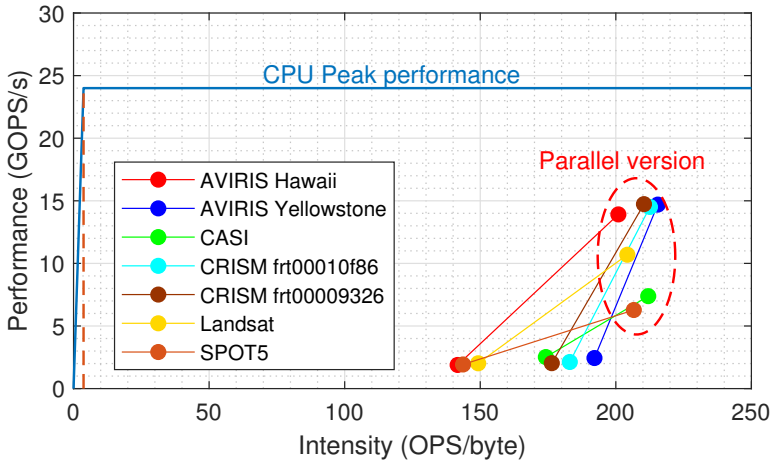


Figure 6.32: Roofline model for the CCSDS 123 sample adaptive encoder parallelized in the CPU.

6.5 Throughput and Energy-efficiency Analysis

The Jetson TX2 comes with built-in power monitors capable of measuring six different rails [38]. The tests were executed with a bash script that takes power measurements at a certain frequency. The manufacturer recommends to have a measuring frequency of a second or more since reading internal nodes too frequently, incurs in an excessive amount of energy consumption [38]. However, some tests run in less than one second. In those cases, the sampling frequency is increased and the difference between the two states is subtracted to the power measurements in tests executed with high sampling rates.

To optimize power efficiency, the Jetson TX2 has 5 power modes described in appendix F. The only 2 modes valid for the parallelization are Max-N and Max-P All core modes. The other modes do not have all cores activated. All tests, until this point, were executed in Max-N mode but in [25], it is noted that running in MAX-P All cores mode, brings maximum system performance under less than 15 W. With that information in mind, the best results from both the predictor and the encoder were tested using the 2 power modes. The obtained results are presented in appendix G.

For the predictor, a throughput of 1.038 GSa/s was obtained by AVIRIS Yellowstone running in MAX-N. In terms of efficiency, the best result was attained by running in MAX-P mode for the CASI image (269.436 MSa/s/W). For the encoders, the results are poorer than the predictor because of data dependencies. In the block adaptive encoder, the best results were collected by the Landsat image. In throughput performance (97.022 MSa/s) and efficiency (21.936 MSa/s/W), for MAX-N mode and MAX-P respectively. For sample adaptive encoder it was tested for both GPU and CPU implementations. The GPU version is the only test where better throughput and efficiencies were obtained in

MAX-N mode. The AVIRIS Hawaii reached 23.053 MSa/s of throughput and 5.435 MSa/s/W of efficiency. In the CPU version, the CRISM frt00009326 attained a throughput of 35.017 MSa/s in MAX-N mode and the AVIRIS Hawaii obtained a efficiency of 5.814 MSa/s/W in MAX-P mode.

In order to calculate the overall performance, the predictor and the encoder must be combined. The best result achieved is for the AVIRIS Hawaii image using the block adaptive encoder. Table 6.2 shows the combination using the 2 power modes.

Table 6.2 Overall power performance for AVIRIS Hawaii using the predictor ($P = 0$) and the block adaptive encoder for the 2 power modes. * Higher is better.

Predictor power Mode	**Encoder power mode**	**Power (W)**	**Time (s)**	**Throughput (MSa/s)**	**Efficiency * (MSa/s/W)**
MAX-N	MAX-N	6.276	0.806	87.384	13.925
MAX-N	MAX-P	4.646	0.997	70.641	15.204
MAX-P	MAX-N	6.142	0.816	86.292	14.049
MAX-P	MAX-P	4.555	1.007	69.925	15.353

6.6 Summary

This chapter analyzes the results obtained for the parallelization of the predictor and the encoder. In the predictor, the optimizations are applied progressively and the best combinations of parallelizations are discussed. Then, the top performer parallelization is tested in several images. The encoders are parallelized mainly in CPU using all cores when available. Finally, an energy analysis is performed running the system in two distinct power modes.

7

Conclusions

Contents

The use of low-power graphics processing units (GPUs) brings a new paradigm to the field of multispectral and hyperspectral compression. Even though, not as the efficiency as field-programmable gate arrays (FPGAs), GPUs deliver high throughput rates. The parallelization of the intra-band predictor is quite effective, achieving speedups above 190x. Inter-band prediction did not achieve the same results due to data dependencies. Intra-band prediction always yields a worse compression ratio but consumes less energy to compress the image. If the system is implemented in satellites, using intra-band prediction might be beneficial against inter-band prediction. In this trade-off, intra-band consumes more energy to transmit data to earth-based stations but spends less energy in the compression phase..

The parallelization in the predictor is quite effective for intra-band prediction, breaking the mark of 1 GSa/s (1.038 GSa/s) in terms of throughput performance. The solutions developed for the encoder are satisfactory. By exploring all available compute units, it is possible to attain speedups of 10 times taking less than 1 second to encode hundreds of MBs of data. The overall system, in terms of efficiency, is the best result found in the literature, breaking the record for the GPU-implemented systems, achieving 15.353 MSa/s/W.

When processing data on GPU, the performance achieved is quite sensible to the dimensions of the image. These devices would benefit if the sensor had dimensions that were power of two. In the future, the manufacturers should have this detail in mind when designing new sensors.

The compact reconnaissance imaging spectrometer for mars (CRISM) images are the largest image the Jetson TX2 can handle. The board struggles with the image occupying all the available memory. For larger images, the system must have more memory available.

To conclude, the gap between FPGAs and GPUs is closing in terms of efficiency.However, FPGAs still dominate the best solutions in the market. If we relax power constrains in the system design, GPUs can take advantage due to lower cost and development effort, which significantly increases competitiveness.

7.1 Future work

The results obtained are very positive and encourage to continue investigating this subject. Although good performance has been observed, there are some aspects that can be further studied, namely:

- Expand the kernel to other parameters in the predictor, namely, the full mode and neighbourhood-oriented local sum;

- Parallelize the weights kernel in the central processing unit (CPU) cores, instead of the GPU;
- Expand to the newly issued low-complexity and near-lossless standard consultative committee for space data systems (CCSDS)-123.0-B-2, which expands the predictor, and contemplates a new hybrid encoder;
- Testing the developed system in other low-power GPUs.

Bibliography

[1] A. Martins, "A Low-Power Parallel GPU Mulispectral and Hyperspectral Lossless Compressor," M.Eng. Thesis, University of Coimbra, Portugal, 2017.

[2] R. Brown, J. G. Paine, K. Saylam, T. A. Tremblay, J. R. Andrews, and A. Averett, "Mangrove monitoring using airborne vnir in the espiritu santo bay area, central texas coast," 09 2016.

[3] O. Arslan, O. Akyürek, and S. Kaya, "A comparative analysis of classification methods for hyperspectral images generated with conventional dimension reduction methods," *Turkish Journal of Electrical Engineering & Computer Sciences*, vol. 25, pp. 58–72, Jan. 2017.

[4] L. Santos, A. Gomez, and R. Sarmiento, "Implementation of ccsds standards for lossless multispectral and hyperspectral satellite image compression," *IEEE Transactions on Aerospace and Electronic Systems*, pp. 1–1, 2019.

[5] Consultative Committee for Space Data Systems. CCSDS 123.0-B-1 Lossless Multispectral & Hyperspectral Image Compression . Internet: https://public.ccsds.org/Pubs/123x0b1ec1s.pdf, May, 2012, [Apr. 11, 2019].

[6] C. C. for Space Data Systems, "Ccsds 121.0-b-2 lossless data compression," internet: https://public.ccsds.org/Pubs/121x0b2ec1.pdf, May, 2012, [May. 7, 2019].

[7] Nvidia, "CUDA C Programming Guide," internet: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Mar. 26, 2019, [Apr. 24, 2019].

[8] ——, "DATA SHEET, NVIDIA Jetson TX2 Series System-on-Module," internet: https://developer.nvidia.com/embedded/dlc/jetson-tx2-module-datasheet, [Jan. 22, 2019].

[9] D. Báscones, C. González, and D. Mozos, "Parallel Implementation of the CCSDS 1.2.3 Standard for Hyperspectral Lossless Compression," *Journal of Remote Sensing*, vol. 7, 09 2017.

[10] Nvidia, "Power Management for Jetson TX2 Series Devices," internet: https://docs.nvidia.com/jetson/l4t/index.html#page/Tegra%2520Linux%2520Driver%2520Package%2520Development%2520Guide%2Fpower_management_tx2_32.html%23wwpID0E0VN0HA, Jul. 5, 2019, [Sep. 2, 2019].

[11] B. Huang and A. Plaza, "High-performance computing in remote sensing," *Proceedings of SPIE - The International Society for Optical Engineering*, 10 2011.

[12] L. Santos, E. Magli, R. Vitulli, J. F. López, and R. Sarmiento, "Highly-parallel gpu architecture for lossy hyperspectral image compression," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 6, no. 2, pp. 670–681, April 2013.

[13] L. Santos, L. Berrojo, J. Moreno, J. F. López, and R. Sarmiento, "Multispectral and Hyperspectral Lossless Compressor for Space Applications (HyLoC): A Low-Complexity FPGA Implementation of the CCSDS 123 Standard," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 9, no. 2, pp. 757–770, Feb 2016.

[14] Consultative Committee for Space Data Systems, "CCSDS 123.2-G-1 Lossless Multispectral & Hyperspectral Image Compression ," internet: https://public.ccsds.org/Pubs/120x2g1.pdf, Dec, 2015, [Apr. 24, 2019].

[15] R. L. Davidson and C. P. Bridges, "GPU accelerated multispectral EO imagery optimised CCSDS-123 lossless compression implementation," in *2017 IEEE Aerospace Conference*, March 2017, pp. 1–12.

[16] GISGeography, "Multispectral vs Hyperspectral Imagery Explained ," internet: https://gisgeography.com/multispectral-vs-hyperspectral-imagery-explained/, Apr. 25, 2019, [Apr. 29, 2019].

[17] P. Ghamisi, J. Plaza, Y. Chen, J. Li, and A. J. Plaza, "Advanced spectral classifiers for hyperspectral images: A review," *IEEE Geoscience and Remote Sensing Magazine*, vol. 5, no. 1, pp. 8–32, March 2017.

[18] California Institute of Technology, Jet Propulsion Laboratory, "AVIRIS - Airborne Visible/Infrared Imaging Spectrometer," internet: https://aviris.jpl.nasa.gov/index.html, Jan. 29, 2019, [Apr. 29, 2019].

[19] Consultative Committee for Space Data Systems. LOW-COMPLEXITY LOSSLESS AND NEARLOSSLESS MULTISPECTRAL AND HYPERSPECTRAL

IMAGE COMPRESSION, Pink book. Internet: https://public.ccsds.org/Lists/CCSDS%201230P11/123x0p11.pdf, June, 2018, [Jul. 24, 2019].

[20] A. Rege, "An Introduction to Modern GPU Architecture ," internet: http://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf, [Jan. 22, 2019].

[21] J. Larkin, "GPU Fundamentals," internet: http://www.icl.utk.edu/~luszczek/teaching/courses/fall2016/cosc462/pdf/GPU_Fundamentals.pdf, Nov. 14, 2016, [Jan. 22, 2019].

[22] NVidia, "Nvidia Tesla P100 Whitepapper," internet: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, [Jan. 22, 2019].

[23] P. Micikevicius, "Local Memory and Register Spilling," internet: https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf, 2011, [Aug. 8, 2019].

[24] J. Luitjens, "CUDA Pro Tip: Increase Performance with Vectorized Memory Access," internet: https://devblogs.nvidia.com/cuda-pro-tip-increase-performance-with-vectorized-memory-access/, Dec. 4, 2013, [Aug. 8, 2019].

[25] D. Franklin, "NVIDIA Developer Blog: NVIDIA Jetson TX2 Delivers Twice the Intelligence to the Edge," internet: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/, Mar. 7, 2017, [Dec. 28, 2018].

[26] B. Hopson, K. Benkrid, D. Keymeulen, and N. Aranki, "Real-time CCSDS lossless adaptive hyperspectral image compression on parallel GPGPU amp; multicore processor systems," in *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, June 2012, pp. 107–114.

[27] D. Keymeulen, N. Aranki, B. Hopson, A. Kiely, M. Klimesh, and K. Benkrid, "Gpu lossless hyperspectral data compression system for space applications," in *2012 IEEE Aerospace Conference*, March 2012, pp. 1–9.

[28] R. L. Davidson and C. P. Bridges, "Adaptive multispectral gpu accelerated architecture for earth observation satellites," in *2016 IEEE International Conference on Imaging Systems and Techniques (IST)*, Oct 2016, pp. 117–122.

[29] ——, "Error Resilient GPU Accelerated Image Processing for Space Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 1990–2003, Sep. 2018.

[30] N. Aranki, D. Keymeulen, A. Bakhshi, and M. Klimesh, "Hardware Implementation of Lossless Adaptive and Scalable Hyperspectral Data Compression for Space," in *2009 NASA/ESA Conference on Adaptive Hardware and Systems*, July 2009, pp. 315–322.

[31] D. Keymeulen, N. Aranki, A. Bakhshi, H. Luong, C. Sarture, and D. Dolman, "Airborne demonstration of fpga implementation of fast lossless hyperspectral data compression system," in *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, July 2014, pp. 278–284.

[32] A. Tsigkanos, N. Kranitis, G. A. Theodorou, and A. Paschalis, "A 3.3 gbps ccsds 123.0-b-1 multispectral & hyperspectral image compression hardware accelerator on a space-grade sram fpga," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–1, 2018.

[33] J. Fjeldtvedt, M. Orlandić, and T. A. Johansen, "An efficient real-time fpga implementation of the ccsds-123 compression standard for hyperspectral images," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 11, no. 10, pp. 3841–3852, Oct 2018.

[34] M. Klimesh, "Low-Complexity Lossless Compression of Hyperspectral Imagery via Adaptive Filtering," *Jet Propulsion Laboratory (JPL), California, USA, The Interplanetary Network Progress Report*, vol. 42-163, pp. 1–10, 2015.

[35] A. Abrardo, M. Barni, and E. Magli, "Low-complexity predictive lossy compression of hyperspectral and ultraspectral images," in *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2011, pp. 797–800.

[36] L. M. V. Pereira, D. A. Santos, C. A. Zeferino, and D. R. Melo, "A low-cost hardware accelerator for ccsds 123 predictor in fpga," in *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2019, pp. 1–5.

[37] O. Ferraz, "Thesis results ," internet: https://github.com/oscarferraz/thesis_results, Aug. 29, 2019, [Aug. 29, 2019].

[38] Nvidia, "NVIDIA Jetson TX2: Thermal Design Guide," internet: http://developer.nvidia.com/embedded/dlc/jetson-tx2-thermal-design-guide, May. 1, 2017, [Dec. 30, 2018].

[39] E. S. Agency, "European Space Agency Public License – v2.0," internet: https://amstel.estec.esa.int/tecedm/misc/ESA_OSS_license.html, [Aug. 21, 2019].

[40] L. Fossati, "CCSDS 123 Multispectral Hyperspectral Image Compression upcoming standard," internet: https://amstel.estec.esa.int/tecedm/misc/ESA_OSS_license.html, [Aug. 21, 2019].

[41] C. C. for Space Data Systems, "123.0-B-Info TestData," internet: https://cwe.ccsds.org/sls/docs/Forms/AllItems.aspx?RootFolder=%2Fsls%2Fdocs%2FSLS-DC%2F123%2E0-B-Info%2FTestData&View=%7BAE8FB44C-E80A-42CF-8558-FB495ABB675F%7D&, [Aug. 21, 2019].

[42] M. Clark, "CUDA Pro Tip: Kepler Texture Objects Improve Performance and Flexibility," internet: https://devblogs.nvidia.com/cuda-pro-tip-kepler-texture-objects-improve-performance-and-flexibility/, Feb. 3, 2013, [Aug. 22, 2019].

Appendix A

Table A.1 Table containing the image symbols, adapted from [5]

Symbol	Meaning	Allowed values	Reference
x, y, z	Image coordinate indices	N\A	Page 12
t	Alternate image coordinate index	N\A	2.1
$s_{z,y,x}$, $s(t)$	Image data sample	s_{min} .. s_{max}	Page 12
N_x, N_y, N_z	Image dimensions	N\A	Page 12
D	Image dynamic range in bits	2 .. 16	2.16
s_{min}	Lower sample value limit	-2^{D-1} or 0	2.16
s_{max}	Upper sample value limit	$2^{D-1}-1$ or $2^{D}-1$	2.16
s_{mid}	Mid-range sample value	0 or 2^{D-1}	2.16

Table A.2 Table containing the predictor symbols, adapted from [5]

Symbol	Meaning	Allowed values	Reference
$\sigma_{z,y,x}$	Local sum	N\A	2.2, 2.3
$d_{z,y,x}$	Central local difference	N\A	2.4
$d^{N}_{z,y,x}$, $d^{W}_{z,y,x}$, $d^{NW}_{z,y,x}$	Directional local differences	N\A	2.5, 2.6, 2.7
$\boldsymbol{U_z(t)}$	Local difference vector	N\A	2.8, 2.9
P	Number of spectral bands used for prediction	0 .. 15	2.8, 2.9
$\widehat{d}_z(t)$	Predicted central local difference	N\A	2.14
$\boldsymbol{W_z(t)}$	Weight vector	N\A	2.10, 2.11, 2.13
$\omega_z^{(i)}(t)$, $\omega_z^{(N)}(t)$, $\omega_z^{(W)}(t)$, $\omega_z^{(NW)}(t)$	Weight values	N\A	2.12
Ω	Weight resolution	4 .. 19	2.12
$\boldsymbol{\Lambda_z}$	Weight initialization vector	N\A	2.13
Q	Initial weight value resolution	3 .. Ω+3	2.13
$\widetilde{s_z}(t)$	Scaled predicted sample value	N\A	2.15
R	Register size, in bits	$max(32, D+\Omega+2)$ .. 64	2.15
$\widehat{s_z}(t)$	Predicted sample value	N\A	2.19
$\Delta_z(t)$	Prediction residual	N\A	2.20
$\theta_z(t)$	Difference between $\widetilde{s_z}(t)$ and nearest endpoint s_{min} , s_{max}	N\A	2.21
$\delta_z(t)$	Mapped prediction residual	N\A	2.22
$e_z(t)$	Scaled prediction error	N\A	2.23
$\rho(t)$	Weight update scaling exponent	N\A	2.24
v_{min}	Initial weight update scaling exponent parameters	-6 .. v_{max}	2.24
v_{max}	Final weight update scaling exponent parameters	v_{min} .. 9	2.24
t_{inc}	Weight update scaling exponent change interval	2^4, 2^5 .. 2^{11}	2.24
ω_{min}, ω_{max}	Minimum and maximum weight values	$-2^{\Omega+2}$, $2^{\Omega+2}-1$	2.25

Table A.3 Table containing the encoder symbols, adapted from [5] and [6]

Symbol	Meaning	Allowed values	Reference
$\Sigma_z(t)$	Accumulator	N\A	2.26, 2.28
$\Gamma(t)$	Counter	N\A	2.27, 2.29
γ_0	Initial count exponent	1 .. 8	2.26
k_z'	Accumulator initialization parameter	0 .. $D-2$	2.29
γ^*	Rescaling counter size	$max(4, \gamma_0+1)$ .. 9	2.28, 2.29
$u_z(t)$	Unary codeword length	N\A	2.31
$k_z(t)$	Variable length code parameter	N\A	2.30
U_{max}	Unary length limit	8 .. 32	Figure 2.8, Figure 2.9
B	Output word size	1 .. 8	Page 20
J	Block size	8, 16, 32, 64	Page 20
k	Number of least-significant bits	0 .. 29	Page 21
γ	Second extension	N\A	2.32

B

Appendix B

Table B.1 Table for the image metadata header, adapted from [5]

Field	Width (bits)	Description
User-Defined Data	8	The user may assign the value of this field arbitrarily, e.g., to indicate the value of a user-defined index of the image within a sequence of images.
N_x*Size*	16	The value N_x encoded mod 2^{16} as a 16-bit unsigned binary integer.
N_y*Size*	16	The value N_y encoded mod 2^{16} as a 16-bit unsigned binary integer.
N_z*Size*	16	The value N_z encoded mod 2^{16} as a 16-bit unsigned binary integer.
Sample Type	1	'0': image sample values are unsigned integers. '1': image sample values are signed integers.
Reserved	2	This field shall have value '00'.
Dynamic Range	4	The value D encoded mod 2^4 as a 4-bit unsigned binary integer.
Sample Encoding Order	1	'0': samples are encoded in band-interleaved order. '1': samples are encoded in band-sequential order.
Sub-Frame Interleaving Depth	16	When band-interleaved encoding order is used, this field shall contain the value M encoded mod 2^{16} as a 16-bit unsigned binary integer. When band-sequential encoding order is used, this field shall be all 'zeros'.
Reserved	2	This field shall have value '00'.
Output Word Size	3	The value B encoded mod 2^3 as a 3-bit unsigned binary integer.
Entropy Coder Type	1	'0': sample-adaptive entropy coder is used. '1': block-adaptive entropy coder is used.
Reserved	10	This field shall contain all 'zeros'.

Table B.2 Table for the predictor metadata header, adapted from [5]

Field	Width (bits)	Description
Reserved	2	This field shall have value '00'.
Number of Prediction Bands	4	The value P encoded as a 4-bit unsigned binary integer.
Prediction Mode	1	'0': full prediction mode is used. '1': reduced prediction mode is used.
Reserved	1	This field shall have value '00'.
Local Sum Type	1	'0': neighbor-oriented local sums are used. '1': column-oriented local sums are used.
Reserved	1	This field shall have value '00'.
Register Size	6	The value R encoded mod 2^6 as a 6-bit unsigned binary integer.
Weight Component Resolution	4	The value $(\Omega - 4)$ encoded as a 4-bit unsigned binary integer.
Weight Update Scaling Exponent Change Interval	4	The value $(log_2 t_{inc} - 4)$ encoded as a 4-bit unsigned binary integer.
Weight Update Scaling Exponent Initial Parameter	4	The value (v_{min} + 6) encoded as a 4-bit unsigned binary integer.
Weight Update Scaling Exponent Final Parameter	4	The value (v_{max} + 6) encoded as a 4-bit unsigned binary integer.
Reserved	1	This field shall have value '0'.
Weight Initialization Method	1	'0': default weight initialization is used. '1': custom weight initialization is used.
Weight Initialization Table Flag	1	'0': Weight Initialization Table is not included in Predictor Metadata. '1': Weight Initialization Table is included in Predictor Metadata.
Weight Initialization Resolution	5	When the default weight initialization is used, this field shall have value '00000'. Otherwise, this field shall contain the value Q encoded as a 5-bit unsigned binary integer.
Weight Initialization Table (Optional)	(variable)	(See section 5.3.3.2 of [5].)

Table B.3 Metadata header when the sample adaptive entropy coder is used, adapted from [5]

Field	Width (bits)	Description
Unary Length Limit	5	The value U_{max} encoded mod 2^5 as a 5-bit unsigned binary integer.
Rescaling Counter Size	3	The value $(\gamma^* - 4)$ encoded as a 3-bit unsigned binary integer.
Initial Count Exponent	3	The value γ_0 encoded mod 2^3 as a 3-bit unsigned binary integer.
Accumulator Initialization Constant	4	When an accumulator initialization constant K is specified, this field encodes the value of K as a 4-bit unsigned binary integer. Otherwise, this field shall be all 'ones'.
Accumulator Initialization Table Flag	1	'0': Accumulator Initialization Table is not included in Entropy Coder Metadata '1': Accumulator Initialization Table is included in Entropy Coder Metadata.
Accumulator Initialization Table (Optional)	(variable)	(See 5.3.4.2.2 of [5].)

Table B.4 Metadata header when the block adaptive entropy coder is used, adapted from [5]

Field	Width (bits)	Description
Reserved	1	This field shall have value '0'.
Block Size	2	'00': Block size $J = 8$. '01': Block size $J = 16$. '10': Block size $J = 32$. '11': Block size $J = 64$.
Restricted Code Options Flag	1	This field shall have value '1' when $D \leq 4$ and the Restricted set of code options (as defined in section 5.1.2 of [5]) are used. Otherwise, this field shall have value '0'.
Reference Sample Interval	12	Value of r encoded mod 2^{12} as a 12-bit unsigned binary integer.

C

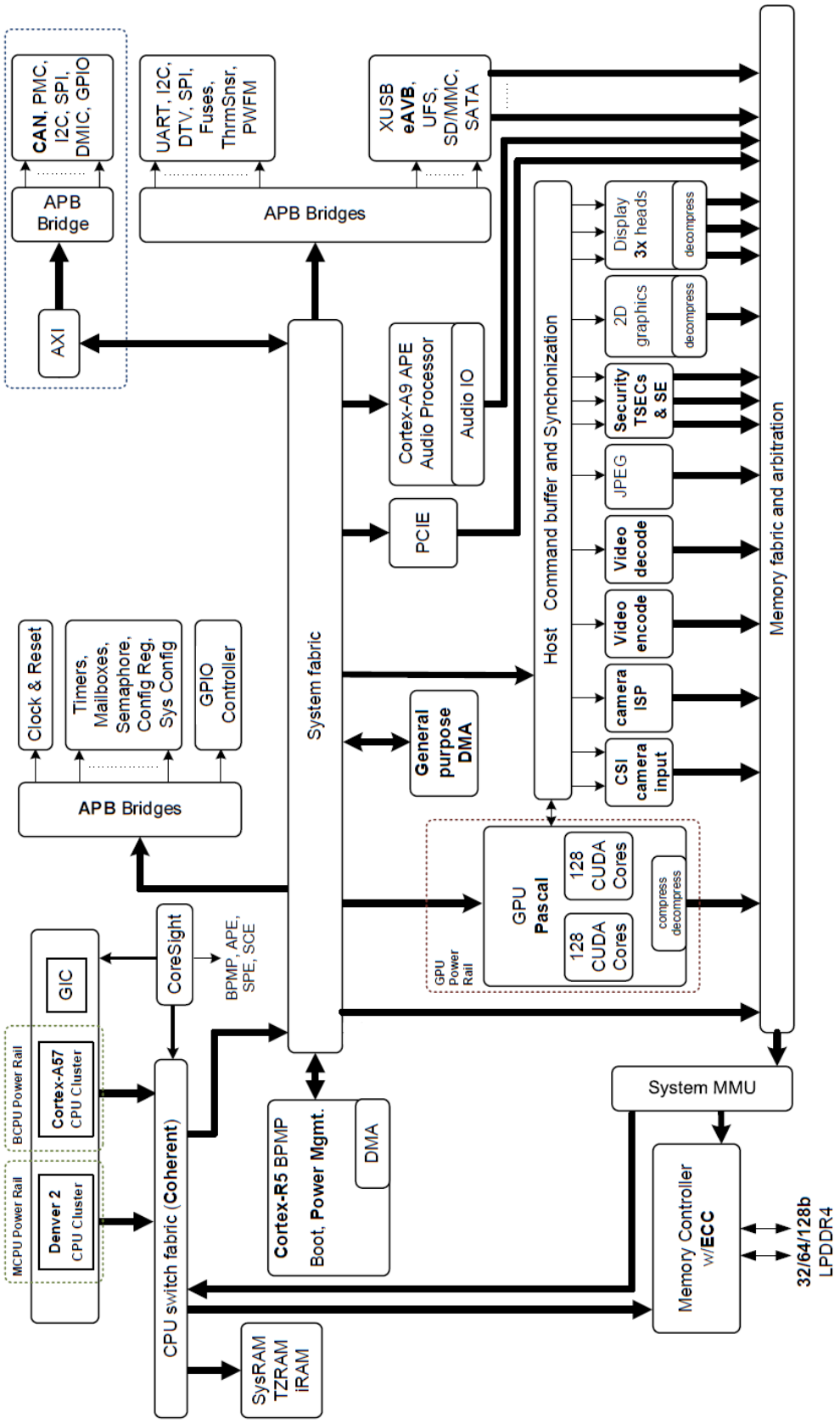
Appendix C

Table C.1 Table for the code option identification key, adapted from [6]

Code Option	Codeword
Zero Block	00000
Second Extension	00001
k =0	0001
k =1	0010
k =2	0011
k =3	0100
k =4	0101
k =5	0110
k =6	0111
k =7	1000
k =8	1001
k =9	1010
k =10	1011
k =11	1100
k =12	1101
k =13	1110
no compression	1111

D

Appendix D


CAN, PMC, I2C, SPI, DMIC, GPIO
APB Bridge
AXI
UART, I2C, DTV, SPI, Fuses, ThrmSnsr, PWFM
APB Bridges
XUSB eAVB, UFS, SD/MMC, SATA
Cortex-A9 APE Audio Processor
Audio IO
PCIE
System fabric
Host Command buffer and Synchonization
Display 3x heads
decompress
2D graphics
decompress
Security TSECs & SE
JPEG
Video decode
Video encode
camera ISP
CSI camera input
Memory fabric and arbitration
Clock & Reset
Timers, Mailboxes, Semaphore, Config Reg, Sys Config
GPIO Controller
APB Bridges
General purpose DMA
GPU Power Rail
GPU Pascal
128 CUDA Cores
128 CUDA Cores
compress decompress
GIC
CoreSight
BPMP, APE, SPE, SCE
BCPU Power Rail
Cortex-A57 CPU Cluster
MCPU Power Rail
Denver 2 CPU Cluster
CPU switch fabric (Coherent)
Cortex-R5 BPMP Boot, Power Mgmt.
DMA
System MMU
Memory Controller w/ECC
32/64/128b LPDDR4
SysRAM TZRAM iRAM

Appendix E

Table E.1 Table with the images dimensions and sizes

Image	Height (rows)	Width (columns)	Bands	Bytes
aviris hawaii	614	512	224	140836864 (134.31 MB)
aviris yellowstone	680	512	224	155975680 (148.75 MB)
crism frt00010f86_07	640	510	545	355776000 (339.29 MB)
crism frt00009326_07	640	510	545	355776000 (339.29 MB)
casi t0477f06-raw	406	1225	72	71618400 (68.30 MB)
Landsat agriculture	1024	1024	6	12582912 (12 MB)
toulouse spot5	1024	1024	3	6291456 (6 MB)

Appendix F

Table F.1 Table with the power modes for the Jetson TX2, from [10]

Mode	Denver 2 activated cores	Denver 2 clock frequency	ARM activated cores	ARM clock frequency	GPU clock frequency	Power budget
Max-N	2	2.0 GHz	4	2.0GHz	1.3 GHz	n/a
Max-Q	0		4	1.2 GHz	0.85 GHz	7.5 W
Max-P Core-All	2	1.4 GHz	4	1.4 GHz	1.12 GHz	15 W
Max-P ARM	0		4	2.0GHz	1.12 GHz	15 W
Max-P Denver	1	2.0 GHz	1	2.0GHz	1.12 GHz	15 W

Appendix G

Table G.1 Power measurements for the predictor ($P = 0$) from the results of Figure 6.11 using the 2 power modes. * Higher is better

Power Mode	Image	Power (W)	Time (ms)	Speed (MSa/s)	Efficiency * (MSa/s/W)	Performance per cycle * (MSa/s/W/GHz)
MAX-N	AVIRIS Hawaii	4.813	67.848	1037.888	215.630	165.869
	AVIRIS Yellowstone	4.836	77.174	1010.545	209.949	160.730
	CASI	4.621	37.984	942.745	204.014	156.934
	CRISM 10f86	5.101	226.646	784.872	153.875	118.366
	CRISM 9326	5.206	225.421	789.136	151.570	116.593
MAX-P	AVIRIS Hawaii	3.610	78.050	902.217	249.906	223..130
	AVIRIS Yellowstone	3.454	83.789	930.765	269.436	240.568
	CASI	3.303	40.192	890.945	269.714	240.816
	CRISM 10f86	3.807	265.499	670.013	175.975	157.120
	CRISM 9326	3.873	253.233	702.467	181.386	161.952

Table G.2 Power measurements for the block adaptive encoder using the 2 power modes. * Higher is better

Power Mode	Image	Power (W)	Time (s)	Speed (MSa/s)	Efficiency * (MSa/s/W)	Performance per cycle * (MSa/s/W/GHz)
MAX-N	AVIRIS Hawaii	6.410	0.738	95.427	14.888	1.943
	AVIRIS Yellowstone	6.637	0.895	87.095	13.122	1.796
	CRISM 10f86	7.805	2.316	76.822	9.842	1.589
	CRISM 9326	7.792	2.345	75.859	9.735	1.607
	Landsat	4.93	0.065	97.022	19.679	3.077
	SPOT5	4.611	0.040	78.996	17.130	3.685
MAX-P	AVIRIS Hawaii	4.634	0.929	75.811	16.361	1.697
	AVIRIS Yellowstone	4.739	1.152	67.716	14.289	1.521
	CRISM 10f86	5.107	3.153	56.411	11.045	1.310
	CRISM 9326	4.972	3.063	58.083	11.683	1.477
	Landsat	3.695	0.078	81.045	21.936	2.865
	SPOT5	3.492	0.052	60.899	17.439	2.892

Table G.3 Power measurements for the sample adaptive encoder using the 2 power modes implemented in the GPU. * Higher is better

Power Mode	Image	Power (W)	Time (s)	Speed (MSa/s)	Efficiency * (MSa/s/W)	Performance per cycle * (MSa/s/W/GHz)
MAX-N	AVIRIS Hawaii	4.242	3.055	23.053	5.435	4.181
	AVIRIS Yellowstone	4.310	3.462	22.527	5.227	4.021
	CASI	3.831	2.835	12.632	3.298	2.537
	CRISM 10f86	3.991	28.676	6.203	1.554	1.196
	CRISM 9326	3.971	28.612	6.217	1.566	1.204
	Landsat	3.916	2.675	2.352	0.601	0.462
	SPOT5	3.656	2.310	1.362	0.372	0.287
MAX-P	AVIRIS Hawaii	3.571	3.778	18.638	5.220	4.660
	AVIRIS Yellowstone	3.606	4.361	17.882	4.959	4.428
	CASI	3.350	3.374	10.613	3.168	2.829
	CRISM 10f86	3.576	31.401	5.665	1.584	1.414
	CRISM 9326	3.576	31.434	5.659	1.583	1.413
	Landsat	2.257	3.026	2.080	0.639	0.570
	SPOT5	3.230	2.711	1.160	0.359	0.321

Table G.4 Power measurements for the sample adaptive encoder using the 2 power modes implemented in the CPU. * Higher is better

Power Mode	Image	Power (W)	Time (s)	Speed (MSa/s)	Efficiency * (MSa/s/W)	Performance per cycle * (MSa/s/W/GHz)
MAX-N	AVIRIS Hawaii	6.482	2.034	34.624	5.342	0.475
	AVIRIS Yellowstone	6.542	2.286	34.106	5.211	0.463
	CASI	6.531	2.056	17.420	2.667	0.250
	CRISM 10f86	6.813	5.213	34.124	5.009	0.452
	CRISM 9326	6.998	5.080	35.017	5.004	0.443
	Landsat	5.263	0.240	26.174	4.974	0.501
	SPOT5	4.964	0.207	15.188	3.060	0.653
MAX-P	AVIRIS Hawaii	4.432	2.732	25.769	5.814	0.739
	AVIRIS Yellowstone	4.526	3.109	25.084	5.542	0.704
	CASI	4.434	2.821	12.693	2.863	0.384
	CRISM 10f86	4.558	7.597	23.416	5.137	0.663
	CRISM 9326	4.554	7.386	24.083	5.287	0.669
	Landsat	3.670	0.346	18.193	4.957	0.713
	SPOT5	3.557	0.287	10.943	3.076	0.938