UNIVERSIDADE Đ COIMBRA

Alexandre Filipe Marcela Martins

A Low-power Parallel GPU Multispectral and Hyperspectral Lossless Compressor

VOLUME 1

Dissertação no âmbito do Mestrado Integrado em Engenharia Eletrotécnica e de Computadores orientada pelo Professor Doutor Gabriel Falcão Paiva Fernandes e apresentada à Faculdade de Ciências e Tecnologia da Universidade de Coimbra.

Setembro de 2018

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

A Low-power Parallel GPU Multispectral and Hyperspectral Lossless Compressor

Alexandre Filipe Marcela Martins

Dissertação para obtenção do Grau de Mestre em
Engenharia Electrotécnica e de Computadores

Orientador:	Prof. Dr. Gabriel Falcão Paiva Fernandes
Co-Orientador:	Prof. Dr. Vítor Manuel Mendes da Silva

Júri

Presidente:	Prof. Dr. Fernando Manuel dos Santos Perdigão
Orientador:	Prof. Dr. Gabriel Falcão Paiva Fernandes
Vogal:	Prof. Dr. Jorge Nuno de Almeida e Sousa Almada Lobo

Setembro 2018

The more I do the more I think, the key with making seems to be working harder and working smarter.
- Alec Steele

Deadlines refine the mind. They remove variables like exotic materials and processes that take too long. The closer the deadline, the more likely you'll start thinking waaay outside the box.
- Adam Savage

Acknowledgments

Sendo sempre difícil a tarefa de um agradecimento quando se sente que esta corre o risco de conter muitas omissões devo, no entanto, deixar aqui expresso o meu reconhecimento a todos os amigos e familiares que, embora não referidos por nome, me ajudaram ou incentivaram por variadas formas.

Em primeiro lugar, agradeço aos meus orientadores, Professor Doutor Vítor Manuel Mendes da Silva e Professor Doutor Gabriel Falcão Paiva Fernandes, pela amabilidade, compreensão, profissionalismo, disponibilidade e por todo o apoio prestado.

Seguidamente gostaria de agradecer ao Instituto de Telecomunicações pela cedência do espaço e material necessário para a realização desta tese.

Aos colegas de laboratório agradeço o apoio e disponibilidade em todos os momentos que precisei. Em especial ao Eduardo Preto pela companhia e camaradagem e pelos momentos de descontração durante este última etapa dos nossos estudos.

À *SQUAD* por todos os momentos bem passados, todo o apoio que existe entre nós, e o esforço que me deram para finalizar esta fase. Um obrigado também à minha colega de BD e amiga Ana Luísa.

Um grande obrigado ao meu amigo e *colega dos canecos* Daniel Freire porque sei que sempre que precisar de alguma coisa estará lá.

Aos meus pais, a quem devo a conclusão desta etapa que não teria sido possível sem o seu esforço, dedicação e apoio.

Por fim, um grande e especial agradecimento à Inês, por todo o apoio, preocupação, sinceridade, por me pressionares a ser cada vez melhor e mais exigente comigo próprio e por me tornares melhor a cada dia que passa.

A todos, muito obrigado!

Abstract

Low power computing is a field ruled by Field-programmable gate arrays (FPGAs) due to their very low power consumptions. But over the years the every day use of mobile devices has pushed for an increase in performance and power efficiency of the Central Processing Units (CPUs) and Graphical Processing Units (GPUs) used to build them.

The main goal of this work was to study the possibility that low power mobile GPUs could be on equal footing with FPGAs when running the Consultative Committee for Space Data Systems (CCSDS) 123 algorithm for compression of Multispectral and Hyperspectral images. The NVIDIA *Tegra K1* chip provides a platform that booth supports Compute Unified Device Architecture (CUDA) programming, and offers a low power consumption, the GPU only consumes an average of 2 Watts.

To speedup the execution of the algorithm, the version developed in this thesis is based on the parallelization of the two main components of the same, the predictor and the encoder. This version is based on a already existing and verified serial solution publish by European Space Agency (ESA). To be able to perform the needed calculations in parallel a study of the algorithm and the way that the compression works was performed. The prediction and the encoding do not have any data dependency between different bands, only inside the same band. So each band can be calculated independently.

The solution was verified comparing all the important function outputs with those from the original code. The final executions times were measured after tunning of the CUDA execution parameters in order to obtain the best results.

The results obtained with this solution presented in this thesis were not outstanding in terms of speed, but despite that, the energy efficiency is very high since the *Tegra K1* GPU is a very power efficient device. The main objective was met despite the less than optimal speed achieved, because the platform demonstrated a good power efficiency.

Keywords

CCSDS 123, CUDA, Parallel Programming, Multispectral, Hyperspectral, Low Power GPU

Resumo

O campo de computação de baixo consumo é dominado por *Field-programmable gate arrays (FPGAs)* devido ao seu baixo consumo energético. mas ao longo dos anos o aumento do uso de dispositivos móveis têm proporcionado o aumento de desempenho e da eficiência energética dos *Central Processing Units (CPUs)* e *Graphical Processing Units (GPUs)* neles usados.

O objetivo principal deste trabalho foi estudar a possibilidade de que GPUs de baixo consumo energético possam estar em pé de igualdade com FPGAs quando executando o algoritmo *Consultative Committee for Space Data Systems (CCSDS)* 123 para compressão de imagens Multiespectrais e Hiperespectrais. O *chip Tegra K1* da *NVIDIA* fornece uma plataforma que suporta programação com *Compute Unified Device Architecture (CUDA)* e também oferece baixo consumo energético. É importante salientar que o GPU apenas consome 2 *Watts*.

Para acelerar a execução do algoritmo, a versão desenvolvida no âmbito desta tese baseia-se na paralelização dos dois blocos mais importantes do mesmo, o preditor e o codificador. Esta versão é baseada numa versão série já existente e validada do algoritmo publicada pela Agência Espacial Europeia (ESA). Para se conseguir realizar todos os cálculos necessários em paralelo, foi feito uma análise do algoritmo e da forma como este executa a compressão. Tanto a predição como a codificação não têm dependências de dados entre bandas diferentes, apenas dentro da mesma banda. Logo, cada banda pode ser calculada independentemente.

Esta solução foi verificada comparando todos os *outputs* das funções importantes, com os mesmos do código original. O tempos de execução finais apenas foram obtidos no fim de fazer afinação dos parâmetros de execução do CUDA de modo a obter os melhores resultados possíveis.

Os resultados obtidos pela solução apresentada nesta tese não foram excecionais em termos de velocidade, mas apesar disso, a eficiência energética da mesma foi alta visto que o GPU do *Tegra K1* é um dispositivo muito eficiente. O objetivo principal foi atingido apesar da velocidade obtida não ser a esperada, uma vez que a plataforma demonstrou uma ótima eficiência energética.

Palavras-Chave

CCSDS 123, CUDA, Programação Paralela, Multiespectral, Hiperespectral, GPU de Baixo Consumo

Contents

List of Figures

List of Tables

List of Acronyms

GPU	Graphical Processing Unit
GPGPU	General-purpose Processing on Graphics Processing Units
CPU	Central Processing Unit
FPGA	Field-programmable gate array
CCSDS	Consultative Committee for Space Data Systems
CUDA	Compute Unified Device Architecture
TDP	thermal design power
SMX	Next Generation Streaming Multiprocessor
SMs	Streaming Multiprocessors
SoC	System on a Chip
SECDED	Single-Error Correct Double-Error Detect
ECC	Error-correcting code
SIMT	Single Instruction Multiple Thread
L4T	Linux for Tegra
BIP	Band Interleaved by Pixel
BSQ	Band-Sequential
ESA	European Space Agency

1

Introduction

1.1 Motivation

Over the years the every day use of mobile devices has pushed for an increase in performance and power efficiency of the Central Processing Units (CPUs) and Graphical Processing Units (GPUs) used to build them. The Tegra GPU series bridges the gap between strictly mobile GPUs (as the one in snapdragon processors) and desktop level GPUs.

This increase in performance and the use of the same architecture as desktop GPUs, enables that this power efficient chips to be used as data processing tools recurring to parallel programming.

Given those enhancements in GPU manufacture bring the power consumptions down, increasing the efficiency (performance per Watt), there is a possibility that mobile GPUs can rival Field-programmable gate arrays (FPGAs) in terms of efficiency in certain tasks.

Since from the information that is readily available FPGAs are the front runners for implementations of the CCSDS 123 Multispectral and Hyperspectral image compression [1, 2], the Tegra platform seems to bring a decent opposition to FPGAs on this field.

1.2 Objectives

This thesis will focus on the study of the possibility that low power mobile GPUs, such as the Tegra K1, can rival FPGAs in their of power efficiency and performance when compared to desktop GPU and FPGA solutions. For this the main objectives are the following:

- Research all the previously proposed solutions present in the literature and already available and tested implementations of the CCSDS 123 algorithm [1];
- Develop a parallel version to speedup the process recurring to CUDA without compromising the process;
- Compare the performance (Msa/s) and efficiency(Msa/s/W) obtained with state-of-the-art solutions.

1.3 Outline

The present thesis is composed of 6 chapters. In the current chapter was made an introduction of the dissertation, which includes the objectives and the motivation to accomplished this work.

In Chapter 2, it will be made a description of the algorithm and an overview of the previous developments in the area using different platforms. In Chapter 3 it will be discussed the development environment used in this work and its GPU architecture, and a small description of how the CUDA model operates. Chapter 4 will focus on the parallel implementation of the algorithm in CUDA. Experimental results are presented in chapter 5 and the conclusions as well as the perspectives of future work, in Chapter 6.

2

Over view of Multispectral and Hyperspectral Image Compression

A Multispectral or Hyperspectral image is one that captures image data within specific wavelength ranges across the electromagnetic spectrum. The information of the spectrum, for each pixel of the image, and the processing of this data, has the porpuse of finding objects, identifying materials, or detecting processes.

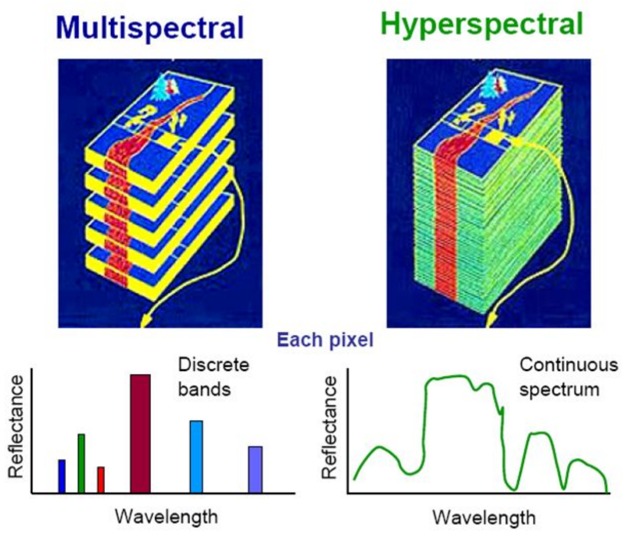


Figure 2.1: Differences between Multispectral and Hyperspectral imaging, adapted from [3].

The difference between Multispectral and Hyperspectral imaging is the number of bands, and how the data is aquired. In Multispectral imaging there is a low number of bands, between 3 and 10 [4] with high bandwidth, representing discrete bands. Hyperspectral imaging has hundreds of narrow bands between 10 and 20 nm each [4] that approximates a continuous spectrum as seen in Figure 2.1. A typical representation of a Hyperspectral image can be seen in Figure 2.2.

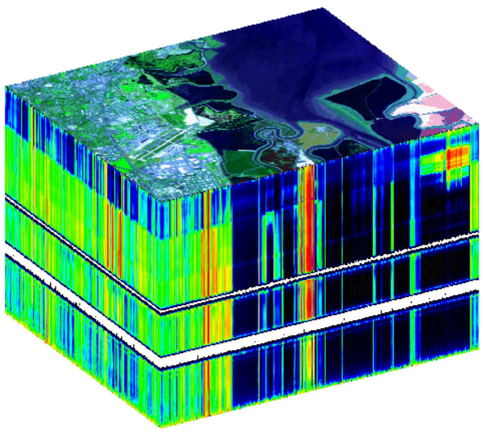


Figure 2.2: Hyperspectral image from the JPL's Airborne Visible/Infrared Imaging Spectrometer (AVIRIS) [5].

Due to the nature of this type of imaging, the sensors produce high volumes of raw data. Most applications of this type of imaging require the transfer of the compressed captured data (e.g. satellites) because of limited onboard storage and downlink bandwidth.

2.1 CCSDS 123

The only solution for this problem is compression, but since all the information is important, the compression must be lossless. The Consultative Committee for Space Data Systems (CCSDS) introduced a standard for lossless Multispectral and Hyperspectral image compression (CCSDS 123 [1]).

This standard describes a lossless compression algorithm that outputs an encoded bitstream from which the image can be completely reconstructed. It has main two functional blocks, namely, a predictor and an encoder as shown in Figure 2.3 [1].

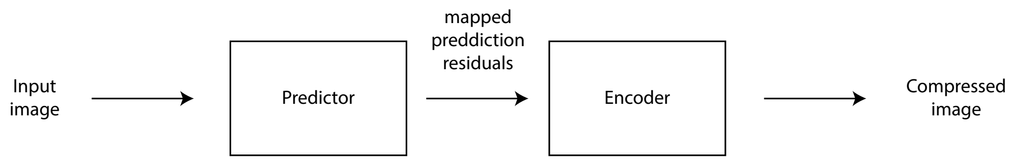


Figure 2.3: Schematic of the CCSDS 123 compressor, adapted from [1].

The predictor is based on the local values of neighboring samples, as seen in Figure **??**. The prediction is performed sequentially. The algorithm can be used with neighbor-oriented or column-oriented local sums, The local sum $\sigma_{z,y,x}$ is calculated with the value of the samples $S_{z,y,x}$ (samples can also be denoted as $S_z(t)$ where z represents the band and $t = x + y \cdot N_x$) as

$$\sigma_{z,y,x} = \begin{cases} S_{z,y,x-1} + S_{z,y-1,x-1} + S_{z,y-1,x} + S_{z,y-1,x+1}, & y > 0, 0 < x < N_x - 1 \\ 4 \cdot S_{z,y,x-1}, & y = 0, x > 0 \\ 2 \cdot \left(S_{z,y-1,x} + S_{z,y-1,x+1}\right), & x = 0, y > 0 \\ S_{z,y,x-1} + S_{z,y-1,x-1} + 2 \cdot S_{z,y-1,x}, & x > 0, x = N_x - 1, \end{cases} \tag{2.1}$$

for neighbor-oriented. For column-oriented the sum is calculated as

$$\sigma_{z,y,x} = \begin{cases} 4 \cdot S_{z,y-1,x}, & y > 0 \\ 4 \cdot S_{z,y,x-1}, & y = 0, x > 0. \end{cases} \tag{2.2}$$

For the sample at $x = 0, y = 0$ the sum is not defined because it is not used [1].

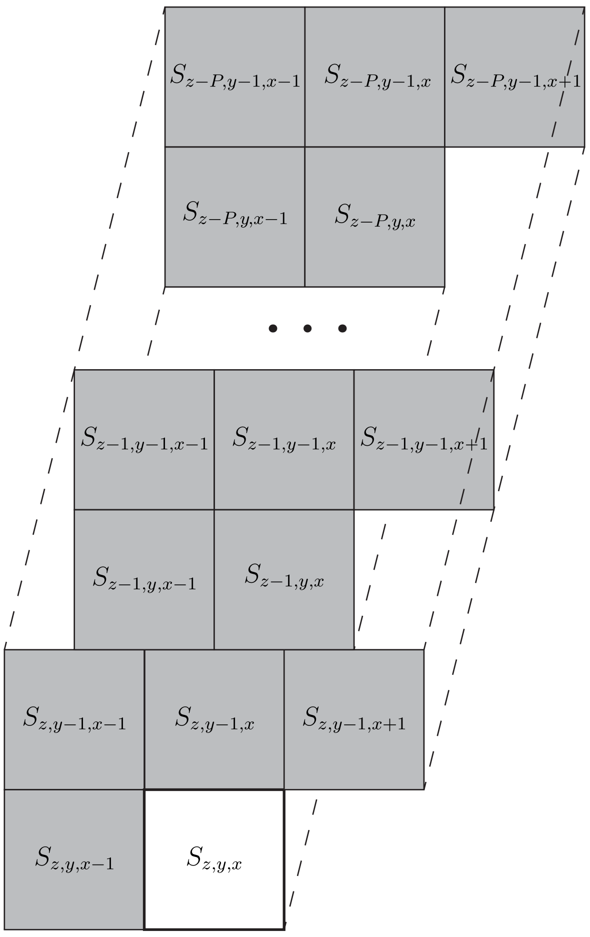


Figure 2.4: Typical Prediction Neighborhood, adapted from [1].

When both x and y are different of zero, the local difference is called central local difference and is defined as

$$d_{z,y,x} = 4 \cdot S_{z,y,x} - \sigma_{z,y,x} \tag{2.3}$$

$$d^{N}_{z,y,x} = \begin{cases} 4 \cdot S_{z,y-1,x} - \sigma_{z,y,x}, & y > 0 \\ 0, & y = 0 \end{cases} \tag{2.4}$$

$$d^{W}_{z,y,x} = \begin{cases} 4 \cdot S_{z,y,x-1} - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4 \cdot S_{z,y-1,x} - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases} \tag{2.5}$$

$$d^{NW}_{z,y,x} = \begin{cases} 4 \cdot S_{z,y-1,x-1} - \sigma_{z,y,x}, & x > 0, y > 0 \\ 4 \cdot S_{z,y-1,x} - \sigma_{z,y,x}, & x = 0, y > 0 \\ 0, & y = 0 \end{cases} \tag{2.6}$$

When $x_1 = x_2$ and $y_1 = y_2$ the difference is central (2.3). When they differ it is used the directional differences (2.4, 2.5, 2.6) ($d^{N}_{z,y,x} d^{W}_{z,y,x} d^{NW}_{z,y,x}$ [1]). The differences are stored in a difference vector $U_z(t)$.

$$U_z(t) = \begin{bmatrix} d_z^N(t) \\ d_z^W(t) \\ d_z^{NW}(t) \\ d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P}(t) \end{bmatrix} \tag{2.7}$$

Each component of the local difference vector ($U_z(t)$) is multiplied by the corresponding weight value (when $t > 0$). The weight vector $W_z(t)$ is then

$$W_z(t) = \begin{bmatrix} \omega_z^N(t) \\ \omega_z^W(t) \\ \omega_z^{NW}(t) \\ \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \vdots \\ \omega_z^{(P)}(t) \end{bmatrix} \tag{2.8}$$

For $t > 0$ the predicted local difference is defined as the inner product of the local difference and weight vectors:

$$\hat{d}_z(t) = W_x^T(t) U_z(t) \tag{2.9}$$

and is then used to calculate the scaled predicted sample value $\tilde{s}(t)$ [1]. The predicted sample value is

$$\hat{s}_z(t) = \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor \tag{2.10}$$

The weights are then updated based on the scaled prediction error

$$e_z(t) = 2S_z(t) - \tilde{s}_z(t) \tag{2.11}$$

The result of the prediction stage is a mapped residual $\delta_z(t)$ which is calculated based on the diference between the sample value and the predicted value.

$$\Delta_z(t) = S_z(t) - \hat{s}_z(t) \tag{2.12}$$

The value to be sent to the encoder is the mapped prediction residual for each sample. The encoder outputs a bitstream containing a header, with the image and compression parameters, so that the image can be decompressed without knowing the original specifications, and a body, with the encoded mapped prediction residuals.

2.2 Previous Developments

Various approaches have been pursued to speedup the CCSDS 123 algorithm [1]. Since a single-threaded implementation can only be so much optimized, parallelization has been the base of these approaches [2, 6–12].

The two main competing platforms are FPGAs and GPUs [2].

2.2.1 FPGA

FPGA-based systems offer flexibility and performance of custom designed hardware. Space applications require hardware with special radiation protection, that is offered by many FPGA boards (e.g. *Xilinx Virtex 5QV* FPGA [6]). FPGAs also offer low power consumption, that results in higher efficiency.

Some proposals are not focused in improving the speed of the compression. In [9] they removed the need for on board storage and low occupancy which resulted in very low power consumption, but the speed of the algorithm dropped. In all [2, 10–12] there was achieved real-time compression, the system discribed in [2] is the only one that does not require external RAM, reducing the power consumption of the system.

2.2.2 OpenMP

OpenMP is an API that facilitates the developing of parallel applications. OpenMP works by dividing iterations of loops between the available threads. It is very easy to implement, but can only run on a single multicore CPU. The latest version, OpenMP 4.0 introduced support for GPU parallelization [13].

The OpenMP proposal in [6] ignores the band-parallelism to get the best performance, using only image segmentation to parallelize the algorithm. They get a substantial speedup compared with [7] that only use band-parallelism.

2.2.3 OpenCL

OpenCL is a C/C++ based API enables the programming of parallel applications that runs in both CPUs and GPUs. The solutions using API can run on more devices than, for example, those using the CUDA API, being that is a royalty-free standard for cross-platform [14].

In [2] they compare head-to-head similar approaches of the algorithm in OpenCL and on a FPGA.

2.2.4 CUDA

CUDA is an API designed by NVIDIA for parallel GPU programming. CUDA is designed to work with NVIDIA GPUs, which means that specific hardware is needed to utilize this API. But since the API is optimized for the hardware, it shows better performance compared with general APIs, such as OpenCL.

Since GPUs are readily available, they are preferred to in atmosphere applications of the algorithm(e.g. UAVs and High-altitude airframes [6]).

In [6] they attaining the faster speeds of all known solutions.

2.2.5 Results

The Table 2.1 shows, in more detail, the results from the previously discussed approaches.

Table 2.1: Results for the CCSDS 1.2.3 algorithm [1], adapted from [2]. N/S stands for not specified. * Higher is better

Platform	Language	Speed (MSa/s)	Power (W)	Efficiency* (MSa/s/W)
V-5QV FX130T [2]	VHDL	179.7	3.04[1]	59.11
V-4 XC2VFX60 [2]	VHDL	116.0	0.90[1]	122.10
V-5QV FX130T [9]	VHDL	11.3	2.34[1]	4.80
RTAX1000S [9]	VHDL	3.5	0.17[1]	20.59
V-4 LX160 [9]	VHDL	11.2	1.49[1]	7.52
V-5 SX50T [10]	VHDL	40.0	0.70[1]	57.14
V-5QV FX130T [11]	N/S	120.0	3.72[1]	32.25
V-5QV FX130T [12]	N/S	55.4	3.31[1]	16.74
V-7 XC7VX690T [2]	VHDL	219.4	5.30[1]	41.40
GT 440 [2]	OpenCL	62.2	<65[2]	0.96
GT 610 [2]	OpenCL	62.6	<29[2]	2.16
i7-6700 [2]	OpenCL	35.0	<65[2]	0.54
GTX 560M [6]	CUDA	321.91	<75[2]	4.29
GTX 560M SLI [6]	CUDA	356.63	<150[2]	2.38
GTX 580 [7]	CUDA	44.85	<244[2]	0.18
GTX 560M [7]	CUDA	36.87	<75[2]	0.49
Tesla C2070 [7]	CUDA	30.09	<238[2]	0.13
Xeon X5690 (12 cores) [7]	OpenMP	19.14	<130[2]	0.15
i7-2760QM (4 cores) [6]	OpenMP	127.89	<45[2]	2.84

[1]The power values were obtained using Xilinx Power Estimator in [2]

[2]This indicates the max thermal design power (TDP) for the platform given by the manufacturer

Most surpass the real-time compression rate of 20 million samples per second (MSa/s) [15]. If there are no power requirements the best option is CUDA, because it delivers the fastest speedup. But if good efficiency is needed, FPGAs are the way to go, due to their low power requirements. The big power requirements by the different platforms show a big difference in the efficiency of each approach.

2.3 Summary

In this chapter we analyzed currently known approaches of the CCSDS 1.2.3 algorithm [1], and the technologies used.

The CUDA API [16] presents the best results, so it will be the one used in this work. It will be developed to use with the NVIDIA Tegra K1 GPU, using their Jetson development board. The approach will use the code provided by ESA in [17], and will be adapted to use CUDA.

3
Low-power GPU Architecture and CUDA

GPUs were originally developed for releasing the main processor from the computationally-intensive task of rendering graphics and images. Soon, GPU hardware manufacturers (e.g. NVIDIA) realized that if the GPU pipeline could be reprogrammed, and could be used to accelerate generic data processing (and not only graphics), a technique later coined General-purpose Processing on Graphics Processing Units (GPGPU). Presently, GPGPU is supported by parallel programming frameworks and technologies such as CUDA [16] and/or OpenCL [14].

3.1 General overview of a GPU architecture

Even though they are both silicon-based microprocessors and the basic building blocks of CPU and GPU architectures share common aspects, there are many differences between them. A CPU consists of a few cores optimized at working on various sequential tasks and conditional tasks, supported by large cache memory and high frequencies of operation. The GPU contains many more cores and is best at focusing all of the capabilities on a single taks that has massive levels of data-parallelism.

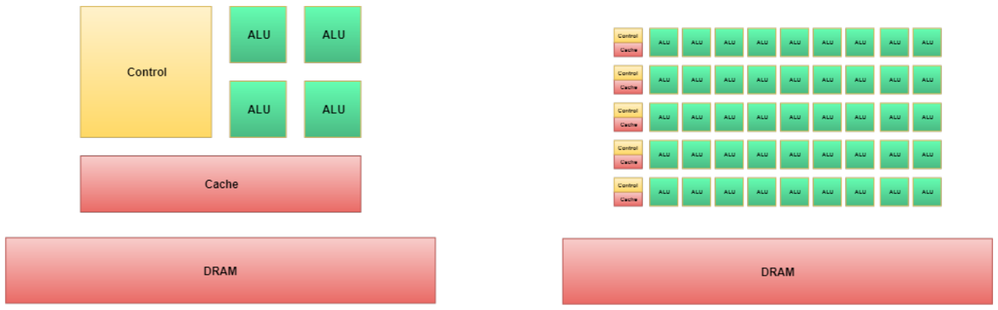


Figure 3.1: Comparison between a generic architectures of a CPU (left) and a GPU (right).

With the advancements required by the demanding complexity of real time graphics generation, e.g in the gaming industry, the gap between CPU and GPU is increasing. The CUDA programming model [16] was introduce by NVIDIA to enable joint operations between CPU and GPU[1].

The GPU used in this work consists of the Tegra K1 from NVIDIA. It was built based on the NVIDIA Kepler architecture, which is the one being addressed under the context of this work.

[1]The CUDA programming model can only be used with NVIDIA GPUs that support this technology.

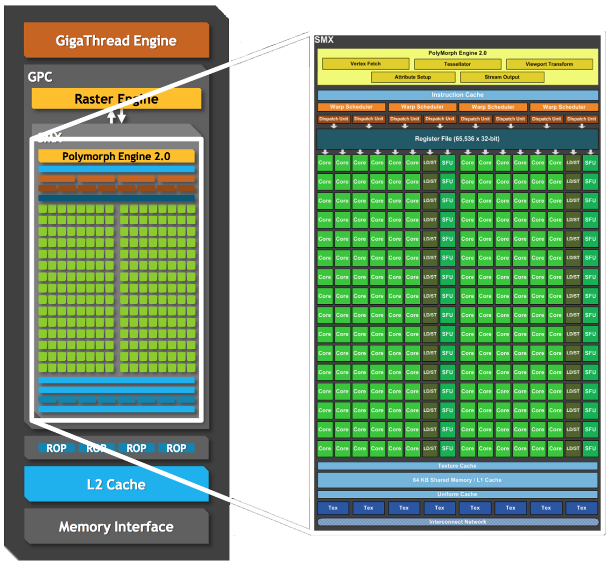


Figure 3.2: Kepler Architecture of the Tegra K1.

3.1.1 Streaming Multiprocessor

NVIDIA GPUs are composed of Streaming Multiprocessors (SMs). Each SM in the Kelper architecture (called Next Generation Streaming Multiprocessor (SMX) [18]) is composed by a control unit, L1 cache, shared memory, a special read-only cache, and 192 CUDA cores. It uses Single Instruction Multiple Thread (SIMT) scheduling to execute each warp, that is a group of adjacent threads within a block. The Tegra K1 chip has a single SMX, as seen in the Figure 3.2, while desktop GPUs using the same architecture can have up to 14 SMs.

3.1.2 Memory Hierarchy

- **SMX Memory**

In each SMX there are 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, as 16 KB of shared memory with 48 KB of L1 cache and even as a 32 KB / 32 KB split between the shared memory and the cache [18]. There is also a 48 KB read-only data cache, that on previous architectures was a texture only cache.

- **GPU Memory**

The whole chip has also a L2 cache with 1536 KB. This cache is responsible to interface between the GPU DRAM and the SMX memory.

All the memory contained in the GPU (L1 cache, shared memory, L2 cache, DRAM and registry files) are protected with by a Single-Error Correct Double-Error Detect (SECDED) Error-correcting code (ECC). The memory interactions can be seen in Figure 3.3. To transfer data to the GPU the CPU can only interact with the GPU DRAM.

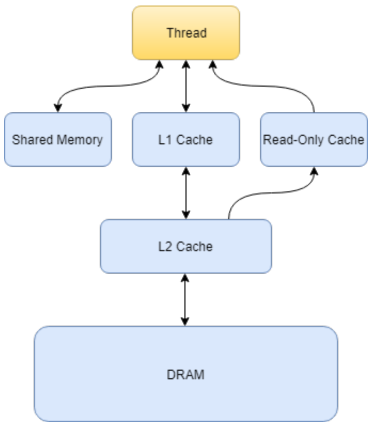


Figure 3.3: Kepler Memory Hierarchy [18].

3.2 The Tegra low-power GPU

The Tegra chip used, is based in the NVIDIA Kepler architecture. The Kepler architecture bridge the gap between the mobile and GeForce GPU architectures. It is a System on a Chip (SoC), that contains both a CPU and a GPU in a single package.

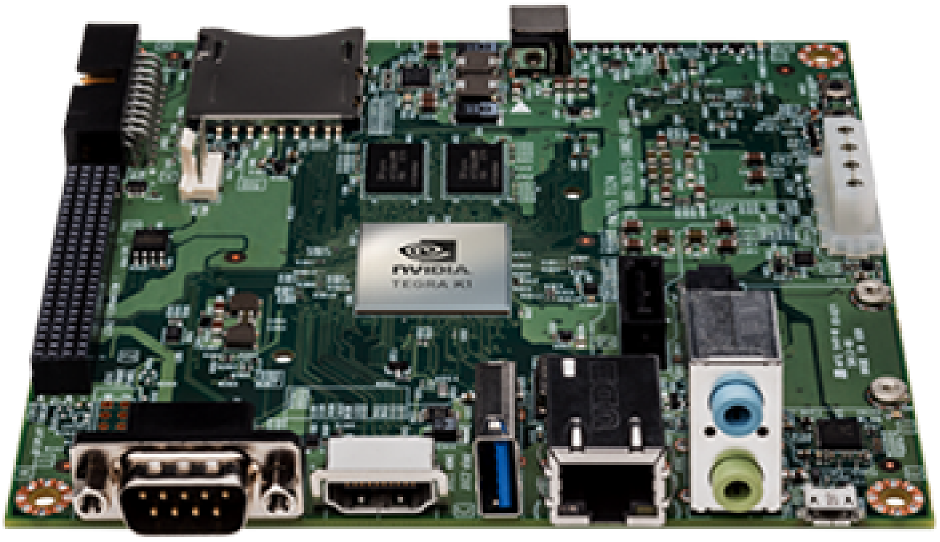


Figure 3.4: Jetson develpment board.

Table 3.1: Tegra K1 Specifications [19].

CPU	
Architecture	NVIDIA® ARM Cortex A15-"R3"
Cores	Quad core + Power Saving Core
Clock	2.3 GHz
GPU	
Architecture	NVIDIA Kepler
CUDA Cores	192
Clock	852 MHz
Memory	
Memory Type	DDR3L and LPDDR3
Max Memory Size	8 GB (with 40-bit address extension)

The CPU is a quad core clocked at 2.3 GHz with another single core used for power saving. The GPU does not have dedicated memory, it shares the RAM with the CPU. As said previously the GPU has a single SM with 192 cuda cores. The diagram of the Tegra K1 chip, and some od its capabilities can be seen in Figure 3.5 And the chip suports a maximum of 8GB of RAM.

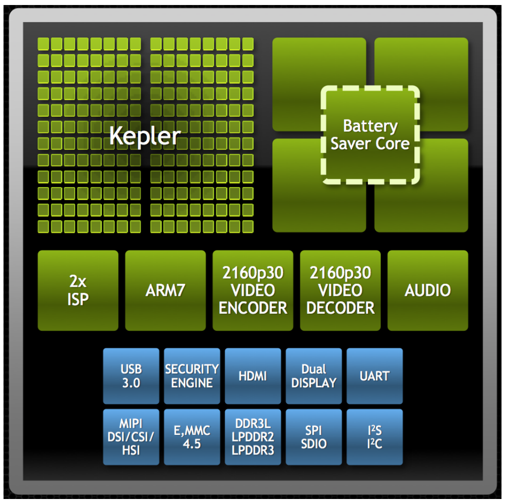


Figure 3.5: Function blocks of the Tegra K1 SoC [20].

The Jetson development board, Figure 3.4, provides a fully fledged system for development kit. Was also used a SSD connected via SATA as a boot drive for the board.

Table 3.2: Jetson TK1 Specifications [21].

Tegra K1 SoC	
CPU	NVIDIA Quad-Core ARM Cortex A15-“R3” CPU
GPU	NVIDIA Kepler with 192 CUDA Cores
Memory	
Memory Type	DDR3L
Memory Size	2 GB x16 Memory with 64-bit Width
Flash Memory	
Memory Type	eMMC 4.51
Memory Size	16 GB
SATA	
Memory Type	SSD
Memory Size	480 GB

The board is build around the Tegra K1 SoC. It has build in flash storage with a special version of ubuntu Linux for Tegra (L4T) made available by NVIDIA. The contents of this storage were copied to the SSD that was used as the primary drive. The board only has 2 GB of RAM that are shared between the CPU and the GPU.

The Tegra K1 GPU has a very low power consumption, only 2 Watts [22]. Comparing with desktop GPUs, e.g. the NVIDIA GTX 580 that is used in one of the approaches referenced in Table 2.1, which has 512 CUDA cores distributed by 16 SMs, running at 1544 MHz but has a power consumption of 244W [23]. The Tegra GPU has half the processor clock, 2.6x less CUDA cores. In terms of memory bandwidth the Tegra K1 GPU has 19 GB/s while the GTX 580 has 192.4 GB/s.

3.3 The CUDA programming model

The CUDA programming model was a solution proposed by NVIDIA to parallel GPU programming. It was made to work with specifically NVIDIA hardware. It is written to be compatible with many popular languages, one of which is C/C++, the programming language used in this work.

CUDA programming allows to write functions, called *kernels*, which are executed by the GPU, all code that is executed in the GPU is called device code. A *kernel* is defined using the code word `__global__` and other functions used in the *kernel* are called device functions and are defined with the code word `__device__`. The code executed by the CPU, that calls the *kernels* is called host code and does not need special definition. When the host code calls a *kernel* it does it using a specific syntax `<<<Nb, Nt>>>` that defines the execution configuration, *Nb* is the number of blocks, and *Nt* is the number of threads per block [24], that produces a grid like the one in Figure 3.6.

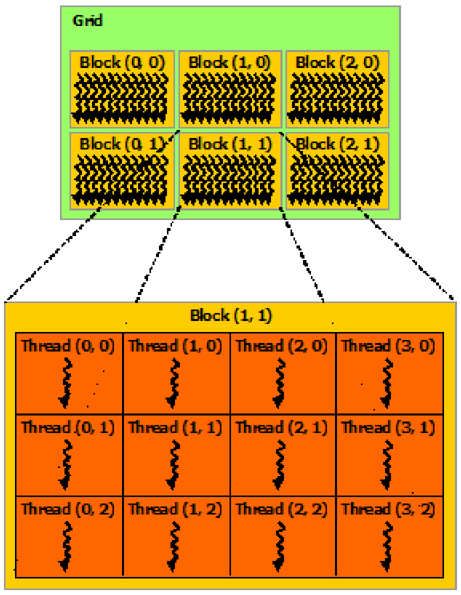


Figure 3.6: Generic *kernel* grid configuration.

3.3.1 Optimization techniques

The biggest challenge in parallel programming is utilizing the capabilities of the hardware in the most efficient way. The following are techniques used to optimize the code and improve the program's performance.

- **Shared memory**

Shared memory is available to all threads in a given SM. The use of shared memory reduces the need to have multiple reads of the same, or locally close, data from the DRAM. It is faster than the GPU DRAM, but is also smaller.

- **Coalescence**

When the treads being executed perform a memory transaction, the device detects if the memory addresses are consecutive, or separated by 32, 64 or 128 bytes. The memory transactions are always performed with fixed sizes, of 32, 64 and 128 bytes. The GPU can coalesce multiple transaction in a single one, decreasing the number of memory accesses needed.

- **Occupancy**

Occupancy is a measure of efficiency, it tells how much of the GPU capabilities the program is using. It is given by:

$$\text{Occupancy}(\%) = \frac{\text{Number of active warps}}{\text{Number of maximum warps}} \times 100$$

for each SM at a certain instant in time. The number of active warps depends not only in the number of threads per block, but also on the requirements for each thread to execute, e.g. number of registers or the amount of local memory used.

The objective is maximizing the occupancy, by utilizing all the processing power available.

- **Arithmetic intensity**

Arithmetic intensity is the ratio between arithmetic operations (operations that do not require off-chip memory access) and memory operations. The efficiency of the program rises with this arithmetic intensity, as memory operation generally require more cycles to be completed, and the arithmetic operations hide the memory access latency.

3.4 Summary

This chapter describes the architecture of GPU devices and the differences between a GPU (device) and CPU (host). The NVIDIA Kepler architecture introduced many improvements in memory management and in the SM architecture. The Jetson development board has a single SM, so only provides 192 CUDA cores.

4

Parallel Multispectral and Hyperspectral Compressor

In this chapter it will be discussed the parallelization strategy of the CCSDS 123 algorithm [1] developed with the CUDA API used in this work. The code used in this work is an adaptation of the code provided by ESA in [17].

To be able to compare the sequential and the parallel algorithm changes to the original code were needed in order to measure the times certain functions take to complete. The main functions to compare are the predictor and the encoder, the two main blocks of the CCSDS 123 algorithm [1], as seen in Figure 2.3.

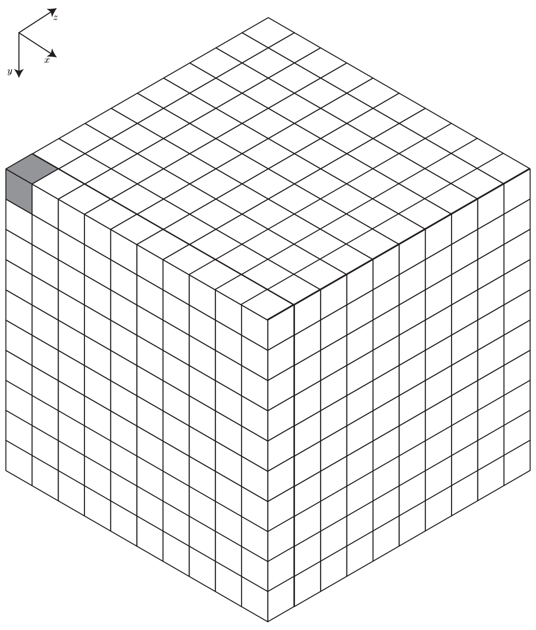


Figure 4.1: Generalization of a Multispectral or Hyperspectral image.

4.1 Workflow

As depicted in Figure 4.2, this application of the CCSDS 123 algorithm [1] presents a pretty standard workflow, it is very similar to Figure 2.3. The blocks focused in this work are the ones that can be parallelized. All the other work performed in the program need to be serial, the read and write to files, and the configuration of the workspace.

It begins to populate some structures defined in the code with the arguments specified by the user. These hold the information of the original image `input_feature_t`, the encoder configuration `encoder_config_t` and the predictor configuration `predictor_config_t`. The raw sample data is then read from the file.

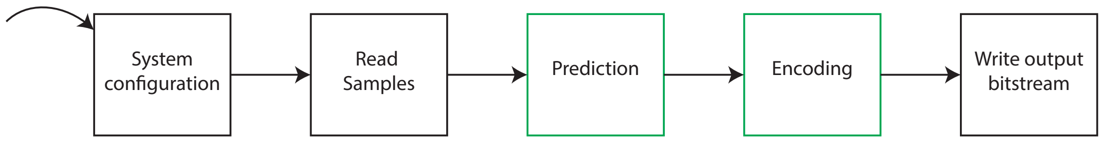


Figure 4.2: Function blocks of the code [17], the ones signaled in green are the ones focused in the work.

The important tasks and the ones that need to be timed are the predictor and the encoder. From the prediction results a array of residuals. The residuals are then interpreted by the encoder, and code words generated for each one. The final bitstream contains an header with the image information and the configuration arguments, and the data portion of that bitstream contains the code words for each encoded residual, producing the final compressed file.

The code was analyzed to discover the best parallelization strategies. To run CUDA code with standard C code the functions in the *.cu* files have to be defined with a `extern ``C''` identifier, because the CUDA compiler treats normal code as C++.

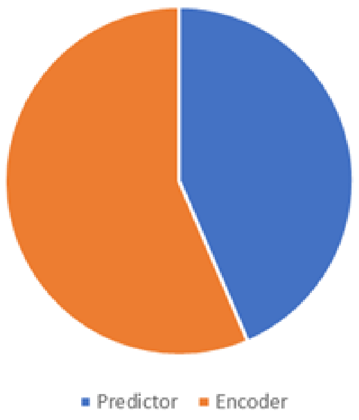


Figure 4.3: Total execution time divided in percentages of the original code [17].

The analysis of the profiling times shows a balanced distribution. Using an average of 20 runs, of two different images, the times divides 55/45 of the total compression time, as seen in Figure 4.3.

4.2 Unified Memory System

During the development of this solution, a limitation of the hardware was found. Since the Jetson development board only has 2GB of RAM, and the RAM is shared between the GPU and CPU, having multiple copies of the same data structures rapidly consumes all the available memory. The algorithm requires the original samples, and a group of neighboring samples, see Figure 2.4, so copying just part of the image would mean having to repeatedly change the contents of the device memory (GPU DRAM). Memory operations are normally slow so another solution was sought.

The solution opted in this work was to use CUDA Unified Memory system [25] using the CUDA function `cudaMallocManaged()` [16].

The Unified Memory in this architecture is more limited, as the only platform that has hardware that supports this technology was introduced in the Pascal architecture, that was first introduced in 2016 [25]. The result of calling `cudaMallocManaged()` is a pointer of the required size in the device memory in the same way as the `cudaMalloc()` in CUDA but this pointer is also accessible by the CPU. In Kepler the GPU driver, since there is no specific hardware to handle this technology it is handled by the software, handles the page table entries for all the pages involved in the allocation [25]. A page fault is generated when the CPU writes to this memory space, and the GPU driver migrates the page to the device memory, where it resides.

On the Jetson development board, used in this work, this presents a form of acceleration, since physically the memory is the same, and memory copies are slow. With this the 2GB of RAM are enough to have all the required data allocated.

4.3 Predictor

The predictor receives the raw sample data and calculates the residuals for each sample. The prediction of the residuals is independent between different bands, but is dependent in the same band, so the calculation has to be divided and the different bands can then be calculated in parallel.

To guarantee that every row is predicted in sequence, since there are data dependencies in the same band, the *kernel* is launched N_y times. The *kernel* calculates N_x samples for each band, and receives the number of the row y that is needed to predict. The index z of the band is calculated with the CUDA execution parameters, the thread and block indexes.

This solution keeps the loop for the x inside the *kernel* to guarantee ordered execution. There is the need to allocate memory so that each band has its own weights vector.

The wrapper that permits the *kernel* to be executed by the regular C code, takes care of the memory allocation and configuration of the CUDA execution.

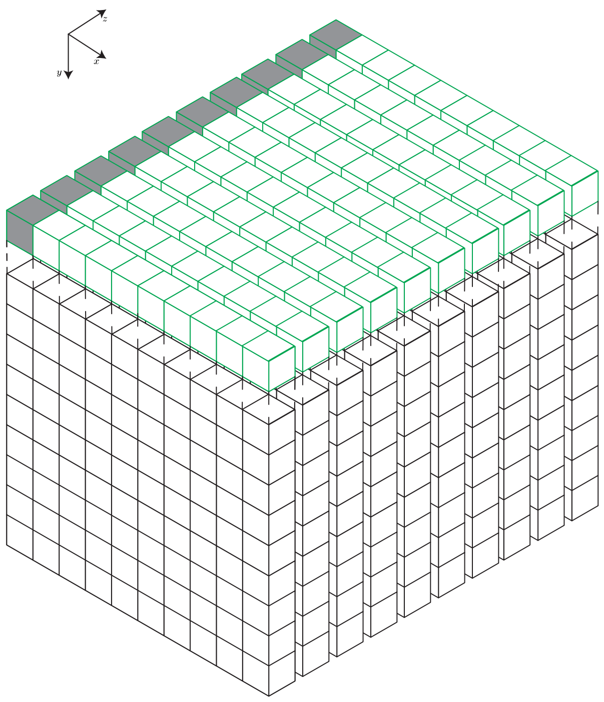


Figure 4.4: Parallel prediction of a Multispectral or Hyperspectral image.

The Figure 4.4 shows how the image is divided for the parallel prediction. The pixels in gray show the group of pixels that are predicted at the same time. The green pixels are the ones that are going to be calculated in the current *kernel* launch, in the next *kernel* launch the *y* index is incremented, and the next batch of lines is calculated.

4.4 Encoder

Having calculated all the residuals they need to be encoded and written to a file. The entropy encoder data dependencies work the same way as in the predictor, the first sample of each band is encoded totally, and the rest of that band is encoded recurring to a *counter* and an *accumulator* that get updated after each pixel of the band is encoded.

For this work there was only parallelized the sample adaptive encoder in Band-Sequential (BSQ) order.

Because of how the statistics are updated the encoding of each different band needs to be sequential. But the codes for each band can be calculated in parallel. There were implemented two methods, the first one, more simpler, just calculates everything needed to write the code words to the bitstream in parallel, and then writes each code word sequentially, maintaining great part of the sequential code, referenced as *PP* (Partially Parallelized) in the results.

And second the solution was, using the same order as the predictor depicted in Figure

4.4, calculate all the codes for each sample. Since the codes for each have variable lengths, there is a need to store each of their sizes. Having the sizes, the offset of each code can be calculated, and the codes can be packed in the final compressed bitstream in parallel. Nothing guarantees that there is not multiple codes that are going to be stored in the same machine word of the output bitstream so, to guarantee that two threads do not try to write to the same word at the same time, each thread writes the code for a band, like this there are $N_x \times N_y$ samples separating the position where each thread is writing to the output bitstream, referenced as *FP* (Fully Parallelized) in the results.

There was also tested a solution with the serial encoder, referenced as *S* (Serial).

4.5 Tests and result analysis

In order to verify that the alterations do not introduce errors in the calculation the mapped residuals and the final bitstream are stored to compare. The solution used the *Input image* to read all the raw data, and produces a file for the *mapped prediction residuals* and the *Compressed image* that contains the final compressed bitstream, as seen in Figure 4.5.

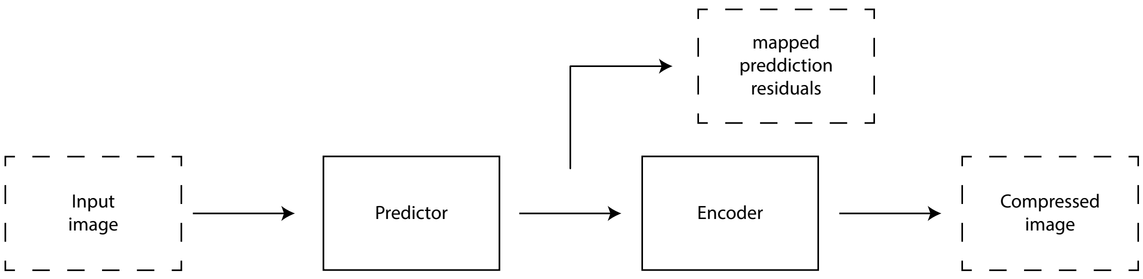


Figure 4.5: Schematic of the CCSDS 123 compressor, adapted from [1]. The dashed boxes represent the files involved in the compression.

The times of execution measured, were all obtained used the same measure. Using the function `clock_gettime()` with the monotonic clock, to get the start and stop time of the functions, and then calculate the difference to get the times.

There was developed two *Python* scripts to extract the results, and afterwards analyze them.

The first script run the application 20 times in a row, and extracts the times from the *stdout* and stores them in a file. The script uses the subprocess package to run the command, and return the program's output.

The second one reads the times from the file, and calculates the average and standard deviation from all the runs. It also presents the best and worst of the runs. It uses the *numpy* package to calculate the average and standard deviation of the sampled runs. Then grabs the minimum and maximum efficiency to find the worst and best runs, respectively.

To analyze visually the images used in this work, the originals, and after the compression/decompression process, in order to guarantee that there are no visual differences in the images, it was used a *Python* package called spectral (*SPy*) [26]. This package permits loading the raw data, and display individual bands as RGB images.

All the results to be analyzed in the next chapter were obtained by these methods.

4.6 Summary

This chapter described the parallelization strategy of the CCSDS 123 algorithm [1] used to develop a CUDA version of the original code [17]. The main differences are to the two main blocks of the algorithm, the predictor and encoder. There was also changes how the memory is allocated to use the GPU to perform calculations.

All the times were measured at the same point in the execution, all using the C function `clock_gettime()`, so that the times can be compared to analyze the performance gain.

In the next chapter the execution times will be compared, and also all the files produced by the program to verify the functionality of this solution.

5 Experimental Results

This chapter presents the results obtained during the development of this work. Before analyzing the final speedup obtained it is needed to verify that the solution produced in this work gives the intended output expected by any decompressor of the CCSDS 123 algorithm [1].

The main image used to analyze the results will be the *hawaii_f011020t01p03r05_sc01* uncalibrated, from the AVIRIS sensor available in [27], since that is the most popular image, and the results can be compared with those presented in previous works, shown in Table 2.1, but other were also used during developments and to analyze the results.

5.1 Software verification

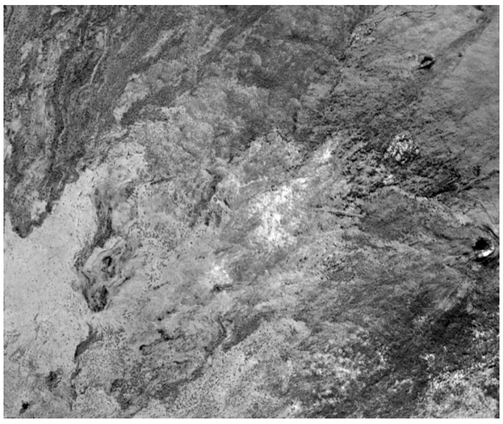
To verify that the program behaves as expected, three data structures have to be compared which are the residuals resulting from the predictor, the final compressed bitstream, and the image after decompression.

The residuals were dumped to a file as *unsigned short int* values (16 bits each) in Band Interleaved by Pixel (BIP) order. Those files were then compared with a file containing the residuals calculated with the original program, using the `diff` UNIX command that compares two files bit by bit. If any of the residuals are not equal to the file with the original residuals, the result of the command will alert that there is a difference between the two files.

The final bitstream, which is stored as the primary output of the program, is also compared in the same way as the residuals. As said previously the compressed bitstream is composed by an header that contains the image information and the configuration used in the predictor and the encoder. The header never differs, because the configuration used to run the algorithm were maintained so that the results could be compared, any difference between the bitstreams will occur in the body of the bitstream.

The two comparisons had positive results which indicates that all the calculations were performed correctly. This version of the compressor did not alter the expected outcome.

Since all the tests were passed, the decompressed image has to be the exactly equal to the original image. To test this first, the previous method is used, to compare the files bit by bit, which as the previous tests gave a positive result.


(a) Original Image

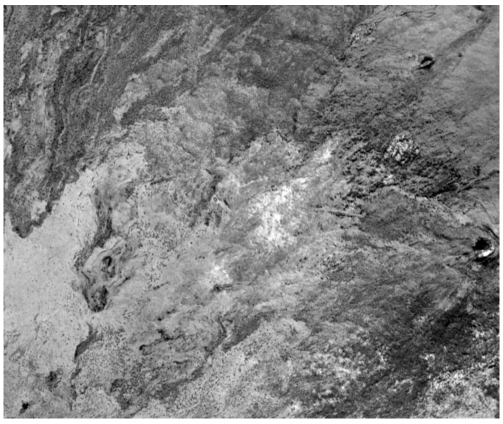
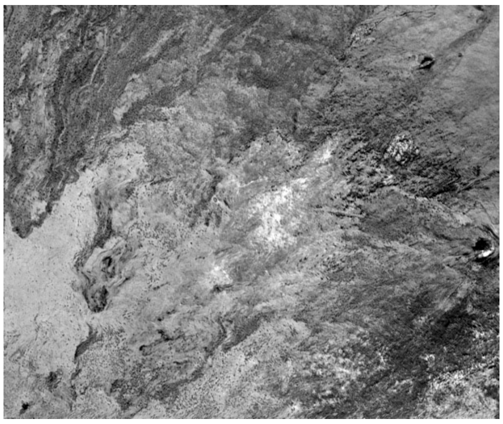

(b) Image after compression and decompression

Figure 5.1: Comparison of the images before and after running the CCSDS 123 algorithm [1] implemented by ESA [17].

The visual comparison only serves to verify that every thing is working in the correct way, since is the less precise method. But is the only discernible way since it shows the user that the process is resulting in the intended output.


(a) Original Image

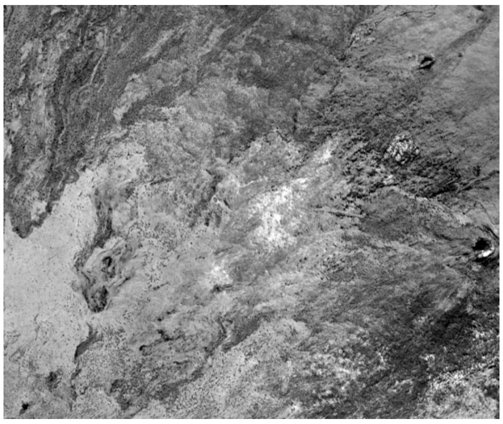

(b) Image after compression and decompression

Figure 5.2: Comparison of the images before and after running the CCSDS 123 algorithm [1] resulting of this work.

As it can be seen comparing Figure 5.1(b) with Figure 5.2(b), which are the resulting images from running the algorithm, they are the same, since the method is lossless there is no quality loss introduced to the image, and by either method the final image is equal to the input image.

As the functionality of the program is verified, the execution times can now be analyzed and compared.

5.2 Parameter tunning

The tunning of the CUDA execution parameters presents another way to improve the execution times of the program. This parameter consist on the number of blocks, and the threads per block. and since the CUDA execution indexes are used in runtime, the total times the *kernel* is executed needs to be higher than the number of bands, so the threads per block are calculated in the following way.

$$\text{Threads per block} = \left\lceil \frac{\text{Number of bands}}{\text{Number of blocks}} \right\rceil \tag{5.1}$$

To find the combination that produces the fastest execution time, all the sets of Table 5.1 were tested. These are calculated for the AVIRIS Hawaii image that has 224 bands, but the method is the same to all images used:

Table 5.1: Number of threads to be executed by each block depending on the number of total blocks.

Blocks	1	2	4	8	16	32	64	128
Threads per Block	225	113	57	29	15	8	4	2
Total kernel executions[1]	225	226	228	232	240	256	256	256

The execution times obtained are presented in Figure 5.3 with the best time represented by the green rhombus.

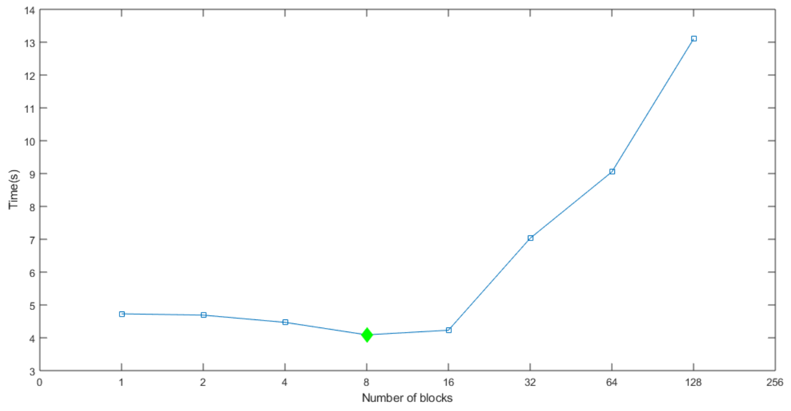


Figure 5.3: Execution times per number of total blocks.

From this image the best configuration found is using 8 blocks and 29 threads per block.

[1]The executions past the number of bands are ignored.

5.3 Execution times

As said in the last chapter all the times presented in the current section will be the averages of 20 runs. The execution times obtained when running the original code [17] were the following.

Table 5.2: Mean execution times of the original code [17] for the *AVIRIS Hawaii* image.

Total time (s)	**Predictor (s)**	**Encoder (s)**	**Speed (MSa/s)**
29.98	13.06	16.92	2.35

Table 5.3: Mean execution times of the original code [17] for the SFSI Mantar image.

Total time (s)	**Predictior (s)**	**Encoder (s)**	**Speed (MSa/s)**
7.33	3.09	4.24	2.27

The objective is to improve upon these times and increase the speed of the program. The main image to compare is the Hawaii from the AVIRIS sensor, but the tests were also performed with the Mantar image from the SFSI sensor [27].

As stated in the precious chapter, there are three versions of the solution developed in this work, *S* (serial), *PP* (partially parallelized) and *FP* (fully parallelized), depending on the version of the encoder that is being used. After running all the versions of the code and extracting the mean times the results were the following.

Table 5.4: Mean execution times for the AVIRIS Hawaii image.

Version	**Total time (s)**	**Predictor (s)**	**Encoder (s)**	**Speed (MSa/s)**
Original	29.98	13.06	16.92	2.35
S	20.94	4.09	16.85	3.36
PP	21.74	4.07	17.66	3.24
FP	50.42	4.07	46.35	1.40

Table 5.5: Mean execution times for the SFSI Mantar image.

Version	**Total time (s)**	**Predictor (s)**	**Encoder (s)**	**Speed (MSa/s)**
Original	7.33	3.09	4.24	2.27
S	5.57	1.37	4.20	2.99
PP	5.83	1.37	4.46	2.86
FP	50.42	1.36	12.42	1.21

As seen in Table 5.4 and 5.5 the best solution is using the original encoder. Any type of attempt to parallelize the encoder results in a slowdown when compared with the original. Since the encoder is a function that produces many memory accesses and not many calculations need to be performed. The bulk of the encoder is storing the code words in memory, since the calculation of these codes are not very intensive. This function has a low arithmetic intensity, and that could be the cause of this slow down.

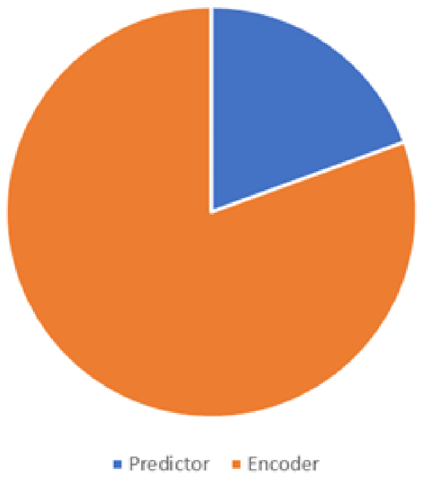


Figure 5.4: Total execution time divided in percentages of the version *S*.

After analyzing the new times, the encoder dropped to approximately 18% of the total time, as seen in Figure 5.4.

The speedup can be calculated for the predictor, and for the total solution, only using the version *S*, since is the one that provides the better results. The speedup (S_{time}) can be calculated by using the following equation.

$$S_{\text{time}}(\%) = \frac{T_{\text{original}}}{T_{\text{final}}} \times 100 \tag{5.2}$$

Table 5.6: Speedup obtained with version *S*.

Image	**Predictor**	**Total**
AVIRIS_Hawaii	319%	143%
SFSI_Mantar	226%	132%

The differences in the speedup between the images is related with their size. The sizes of the *SFSI_Mantar* image are $N_x = 496$, $N_y = 140$, $N_z = 240$, while the *AVIRIS_Hawaii* is much bigger with $N_x = 614$, $N_y = 512$, $N_z = 224$. Despite that the *hawaii* image has less bands, which would mean that is has less calculations that can be performed in parallel, the margin provided by having a larger image (the calculation that need to be performed in series) will convert in a bigger improvement in the execution times.

To compare the results obtained in this work with the previous solutions, referred in Table 2.1, the efficiency of the compression of the *AVIRIS_Hawaii* image has to be calculated. This efficiency is calculated with the speed obtained and the chip TDP.

$$\text{Efficiency (MSa/s/W)} = \frac{\text{Speed (MSa/s)}}{\text{Power (W)}} \tag{5.3}$$

Using the best values obtained with the version *S* and an average TDP of <2 W, referenced in the Tegra K1 white paper [22], the efficiency is 1.68 (MSa/s/W).

Table 5.7: Results for the CCSDS 1.2.3 algorithm [1], adapted from [2] including this work, in gray. N/S stands for not specified. * Higher is better.

Platform	Language	Speed (MSa/s)	Power (W)	Efficiency* (MSa/s/W)
V-5QV FX130T [2]	VHDL	179.7	3.04[1]	59.11
V-4 XC2VFX60 [2]	VHDL	116.0	0.90[1]	122.10
V-5QV FX130T [9]	VHDL	11.3	2.34[1]	4.80
RTAX1000S [9]	VHDL	3.5	0.17[1]	20.59
V-4 LX160 [9]	VHDL	11.2	1.49[1]	7.52
V-5 SX50T [10]	VHDL	40.0	0.70[1]	57.14
V-5QV FX130T [11]	N/S	120.0	3.72[1]	32.25
V-5QV FX130T [12]	N/S	55.4	3.31[1]	16.74
V-7 XC7VX690T [2]	VHDL	219.4	5.30[1]	41.40
GT 440 [2]	OpenCL	62.2	<65[2]	0.96
GT 610 [2]	OpenCL	62.6	<29[2]	2.16
i7-6700 [2]	OpenCL	35.0	<65[2]	0.54
GTX 560M [6]	CUDA	321.91	<75[2]	4.29
GTX 560M SLI [6]	CUDA	356.63	<150[2]	2.38
GTX 580 [7]	CUDA	44.85	<244[2]	0.18
GTX 560M [7]	CUDA	36.87	<75[2]	0.49
Tesla C2070 [7]	CUDA	30.09	<238[2]	0.13
Tegra K1	CUDA	3.36	<2[2]	1.68
Xeon X5690 (12 cores) [7]	OpenMP	19.14	<130[2]	0.15
i7-2760QM (4 cores) [6]	OpenMP	127.89	<45[2]	2.84

Comparing with results from previous solutions, it can be seen that even though the speeds obtained are not outstanding, the throughput per power (efficiency) is comparable with the best GPU solutions.

5.4 Summary

In this chapter the experimental results obtained were presented. First the validation of the changes was performed and presented, to show that the parallel version of the algorithm does not errors to the calculations performed. The tunning of the CUDA execution

[1]The power values were obtained using Xilinx Power Estimator in [2]

[2]This indicates the max TDP for the platform given by the manufacturer

parameters was performed in order to obtain the best executions times that were possible. Then those execution times were shown and analyzed, and were concluded with the calculation of the speedup and efficiency obtained with the parallelization of the algorithm.

The next chapter will present the conclusions and future work perspectives.

6 Conclusions

The main goal of this work was to study the possibility that low power mobile GPUs could be on equal footing with FPGAs in terms of speed and power efficiency when running the CCSDS 123 algorithm [1] and compare the performance with other GPU solutions.

The solution developed in this thesis was not outstanding in terms of speed. The speedup obtained on the predictor was 319% which is not insignificant since it has a large computational intensity. The overall speedup was smaller, just 143% because of all the parallel versions of the encoder tested neither of them presented any improvement.

The energy efficiency obtained is very high (comparing with most GPU solutions) since the Tegra K1 GPU is a very low efficient device that only consumes an average of 2 Watts [22].

Table 6.1: Results obtained for the AVIRIS Hawaii Hyperspectral image.

Total time (s)	**Speed (MSa/s)**	**Power (W)**	**Efficiency (MSa/s/W)**
20.94	3.36	<2[1]	1.68

Low power GPUs brings a good power efficiency to the field, but the performance cannot compete with faster solutions as it can be seen in Table 5.7. They bring the ease of GPU programming to the power level of FPGAs.

As stated in the previous chapter, this solution still has the original serial encoder. There is still room to improve the times obtained by exploring other options to parallelize the encoder, being on a GPU, using CUDA [24], or on the CPU, using for example OpenMP [13]. The predictor and the encoder can also be aggregated, and run concurrently, as soon as each row of residuals are available, the code words can be calculated for those samples.

6.1 Future work

The Tegra K1 is one of the older and least powerful mobile chips produced by NVIDIA, there is a large probability that newer chips, e.g. the Tegra X2 used in the Jetson TX2 development board, and the upcoming Jetson Xavier [28]. As they have newer hardware (build using newer and more optimized architectures) they can have a largely better performance. So these platforms can be studied to ascertain if they can be good competitors to FPGAs in the Multispectral and Hyperspectral compression field.

[1]This indicates the max TDP for the platform given by the manufacturer

Bibliography

[1] The Consultative Committee for Space Data Systems, "Lossless Multispectral & Hyperspectral Image Compression CCSDS 120.2-G-1, Blue Book", accessed: February 2018. [Online]. Available: https://public.ccsds.org/Pubs/120x2g1.pdf

[2] D. Báscones, C. González, and D. Mozos, "Parallel implementation of the ccsds 1.2.3 standard for hyperspectral lossless compression", *Remote Sensing*, vol. 9, no. 10, p. 973, 2017. [Online]. Available: http://www.mdpi.com/2072-4292/9/10/973

[3] M. M. Waqar, "“hyperspectral remote sensing”— presentation", accessed: February 2018. [Online]. Available: https://public.ccsds.org/Pubs/120x2g1.pdf

[4] G. Geography, "Multispectral vs hyperspectral imagery explained", accessed: February 2018. [Online]. Available: https://gisgeography.com/multispectral-vs-hyperspectral-imagery-explained/

[5] AVIRIS, "Airborne visible - infrared imaging spectrometer - data", accessed: February 2018. [Online]. Available: https://aviris.jpl.nasa.gov/data/image_cube.html

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

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

[8] R. Davidson and C. Bridges, "Gpu accelerated multispectral eo imagery optimised ccsds-123 lossless compression implementation", in *Aerospace Conference, 2017 IEEE.* IEEE, 2017.

[9] 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, 2016.

[10] 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 *Adaptive Hardware and Systems (AHS), 2014 NASA/ESA Conference on*. IEEE, 2014, pp. 278–284.

[11] G. Theodorou, N. Kranitis, A. Tsigkanos, and A. Paschalis, "High performance ccsds 123.0-b-1 multispectral & hyperspectral image compression implementation on a space-grade sram fpga", in *Proceedings of the 5th International Workshop on On-Board Payload Data Compression, Frascati, Italy*, 2016, pp. 28–29.

[12] G. Lopez, E. Napoli, and A. G. Strollo, "Fpga implementation of the ccsds-123.0-b-1 lossless hyperspectral image compression algorithm prediction stage", in *Circuits & Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on*. IEEE, 2015, pp. 1–4.

[13] The OpenMP Architecture Review Board, "Openmp", accessed: February 2018. [Online]. Available: https://www.openmp.org/

[14] The Khronos Group Inc., "Opencl overview", accessed: February 2018. [Online]. Available: https://www.khronos.org/opencl/

[15] The Consultative Committee for Space Data Systems, "Lossless multispectral & hyperspectral image compression ccsds 123.0-b-1, green book", accessed: February 2018. [Online]. Available: https://public.ccsds.org/Pubs/120x1g2.pdf

[16] NVIDIA, "Cuda runtime api :: Cuda toolkit documentation", accessed: June 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api

[17] European Space Agency, "European space agency public license – v2.0", accessed: February 2018. [Online]. Available: https://amstel.estec.esa.int/tecedm/misc/ESA_OSS_license.html

[18] NVIDIA, "Nvidia kepler gk110 architecture whitepaper", accessed: June 2018. [Online]. Available: https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[19] NVIDIA, “Tegra k1 next-gen mobile processor”, accessed: June 2018. [Online]. Available: http://www.nvidia.com/object/tegra-k1-processor.html

[20] K. Hinum, “Nvidia tegra k1 soc”, accessed: June 2018. [Online]. Available: https://www.notebookcheck.net/NVIDIA-Tegra-K1-SoC.108310.0.html

[21] NVIDIA, “Jetson tk1 embedded development kit”, accessed: June 2018. [Online]. Available: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

[22] NVIDIA, “Nvidia tegra k1 whitepaper”, accessed: June 2018, Page 10. [Online]. Available: https://www.nvidia.com/content/PDF/tegra_white_papers/tegra-K1-whitepaper.pdf

[23] NVIDIA, “Geforce gtx 580 – specifications”, accessed: June 2018. [Online]. Available: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/specifications

[24] NVIDIA, “Programming guide :: Cuda toolkit documentation”, accessed: June 2018. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[25] NVIDIA, “Unified memory for cuda beginners : Nvidia developer blog”, accessed: June 2018. [Online]. Available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/

[26] T. Boggs, “Welcome to spectral python (spy) — spectral python 0.18 documentation”, accessed: February 2018. [Online]. Available: http://www.spectralpython.net/

[27] The Consultative Committee for Space Data Systems, “Test data 123.0-b”, accessed: February 2018. [Online]. Available: https://cwe.ccsds.org/sls/docs/SLS-DC/123.0-B-Info/TestData/

[28] NVIDIA, “Embedded systems developer kits, modules, & sdks – nvidia jetson”, accessed: Jully 2018. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/