Master thesis : NVIDIA Jetson Xavier AGX as multimedia broadcast system

Auteur : Ossohou, Jean-Lorys
Promoteur(s) : Boigelot, Bernard
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en informatique, à finalité spécialisée en "management"
Année académique : 2021-2022
URI/URL : http://hdl.handle.net/2268.2/14460

University of Liège

Civil engineering - Master in computer science

NVIDIA Jetson Xavier AGX as multimedia broadcast system

Master Thesis conducted for obtaining the Master's degree in Computer Science and Engineering

author: Jean-Lorys OSSOHOU *supervisor:* Prof. Bernard BOIGELOT

Academic year 2021-2022

Acknowledgments

Even if a thesis is mainly based on an individual effort, I could benefit from precious help in moments when I needed it.

More precisely, I would like to thank my company supervisor, an outstanding engineer as gifted as kind, Julien JEMINE who knew how to guide me and help me with his extensive skills and knowledge during the meetings we had. I would also like to thank Nicolas RIXHON, a passionate and talented young computer engineer who greatly facilitated my integration into the Deltatec team. And of course the Information Technology (IT) and hardware (HW) teams of Deltatec who were able to provide me with precious services during real technical problems such as the breakdown of my laptop and of the Jetson platform which was the heart of my work.

I would also like to thank the company Deltatec for giving me the chance to realize this Master thesis. More generally, I would like to thank the entire Deltatec team for their warm welcome and the good atmosphere that prevailed. The atmosphere of a company is often underestimated, however, it played a crucial role in my well-being and de facto in the realization of this Master thesis.

Besides, the company context, I would like to thank Prof. Bernard BOIGELOT, my academic supervisor for his availability and his precise help in the conception of the Master thesis report that meets academic expectations.

Finally, I would like to thank my family and friends, especially my brother Jean-Florian OSSOHOU for their support, advice, and for proofreading this work.

Abstract

University of Liège

Civil engineering - Master in computer science

NVIDIA Jetson Xavier AGX as multimedia broadcast system

Jean-Lorys OSSOHOU
Supervised by Prof. Bernard BOIGELOT
May 30, 2022

On the one hand, with the emergence, in particular, of the Internet of Things (IoT) and artificial intelligence, embedded systems are becoming increasingly popular and are used in many applications, including autonomous machines. The use of autonomous machines has revolutionized the industry, with inventions, such as repetitive task robots used in manufacturing and medical sensors in healthcare. Indeed, to name but two advantages, autonomous machines allow for cost efficiency and quality assurance.
On the other hand, the audiovisual world is becoming increasingly demanding, particularly in terms of image quality. Indeed, while 4K resolution is becoming the quality standard, researchers are already setting up increasingly complex systems to broadcast 8K content on a large scale. This implies a furious growth in the amount of data transferred, given the growth in image quality and the numerous users enjoying video content every single day.
However, the internet network responsible for data transfer is a very limited resource unable to support more and more data without problems. Video streaming is governed by a trade-off between video quality, computation, and compression rate. To succeed in providing ever higher image quality, video content must, therefore, be both compressed at never before achieved compression rates thanks to increasingly complex video encoding standards, and processed by increasingly powerful machines.

The hardware, which is at the center of this work, was the NVIDIA Jetson Xavier AGX developer kit. This developer kit allows to build and produce software solutions on a large scale in Jetson Xavier AGX modules, often used in autonomous machines. Therefore, the purpose of this thesis was to provide a broadcasting solution as powerful and portable as possible on an NVIDIA Jetson Xavier AGX module.

This thesis addresses the issue of video streaming by analyzing the different main video formats, video coding standards, and key video compression techniques. Then, this project deals mainly with the computational part of the trade-off, *i.e.*, providing more computing power to accelerate the decompression of encoded data on embedded systems, via well-known and advanced formats such as H.264 or H.265. For the practical part of this thesis, we will first analyze the decoding performance of a solution based on the specific hardware of the AGX module. Then, we will extend this powerful solution to make it as portable as possible by using CPU and then GPU programming. Finally, we will compare the results provided by the different solutions.

Résumé

Université de Liège

Ingénieur civil - Master en sciences informatiques

Utilisation de NVIDIA Jetson Xavier comme système de diffusion multimédia

Jean-Lorys OSSOHOU
Supervisé par Prof. Bernard BOIGELOT
30 mai 2022

D'une part, avec l'émergence, en particulier, de l'*Internet of Things* (IoT) et de l'intelligence artificielle, les systèmes embarqués sont de plus en plus populaires et sont utilisés dans de nombreuses applications, y compris dans le domaine des machines autonomes. L'utilisation de machines autonomes a révolutionné l'industrie avec des inventions telles que les robots à tâches répétitives, pour la fabrication, et les capteurs médicaux, pour les soins de santé. En effet, pour ne citer que deux avantages, les machines autonomes permettent de réduire les coûts et d'assurer la qualité.
D'autre part, le monde de l'audiovisuel devient de plus en plus exigeant, notamment en termes de qualité d'image. En effet, alors que la résolution 4K est en passe de devenir la norme de qualité, les chercheurs mettent déjà en place des systèmes de plus en plus complexes pour diffuser des contenus 8K à grande échelle. Cela implique une croissance furieuse de la quantité de données transférées, compte tenu de l'augmentation de la qualité des images et des nombreux utilisateurs profitant chaque jour de contenu vidéo.
Or, le réseau internet chargé du transfert de données est une ressource très limitée, incapable de supporter, sans aucun problème, une quantité de données toujours plus importante. Le streaming vidéo est régi par un compromis entre la qualité vidéo, le calcul et le taux de compression. Pour réussir à fournir une qualité d'image toujours plus élevée, le contenu vidéo doit donc être à la fois compressé à des taux de compression jamais atteints auparavant, grâce à des normes d'encodage vidéo de plus en plus complexes, et traité par des machines de plus en plus puissantes.

Le matériel, qui est au centre de ce travail, est le kit de développement NVIDIA Jetson Xavier AGX. Ce kit de développement permet de construire et de produire des logiciels à grande échelle dans les modules Jetson Xavier AGX, souvent utilisés dans les machines autonomes. L'objectif de cette thèse était donc de fournir une solution de diffusion, aussi puissante et portable que possible, sur un module NVIDIA Jetson Xavier AGX.

Cette thèse aborde la question du streaming vidéo en analysant les principaux formats de vidéo, les normes d'encodage vidéo et les principales techniques de compression vidéo. Ensuite, ce projet traite principalement de la partie calcul du compromis, *i.e.*, en fournissant plus de puissance de calcul pour accélérer la décompression des données encodées sur les systèmes embarqués via des formats connus et avancés, tels que H.264 ou H.265. Pour la partie pratique de cette thèse, nous analyserons d'abord les performances de décodage d'une solution basée sur le matériel spécifique du module AGX. Ensuite, nous étendrons cette solution puissante pour la rendre aussi portable que possible en utilisant la programmation par CPU puis par GPU. Enfin, nous comparerons les résultats fournis par les différentes solutions.

Contents

Chapter 1

Introduction

Providing an efficient and reliable solution for the distribution of multimedia streams is not an easy task, especially when it comes to providing such a solution for embedded systems. As the quest for higher video quality rages on, the amount of transferred data that grows with quality must be cleverly handled to avoid congestion in the distribution network. Broadcasting world that was unknown to us until then was studied in details in the context of this thesis. Indeed, this thesis deals with efficient decoding solutions on embedded systems, which aim to follow the market. Due to the demand and maturity of the broadcasting sector, one should note that video compression and decompression algorithms are of increasing complexity. Moreover, to reach high performances in embedded systems, we chose to study and manipulate a powerful development kit, namely the **NVIDIA** Jetson Xavier AGX platform. This hardware brought us into the world of embedded systems, a world that was also totally new for us at the beginning of this project. Although the amount of new material to discover and master was not negligible, it clearly motivated us since one of our personal objectives was to acquire a lot of knowledge throughout this thesis.

To carry out this study, we had the chance to collaborate with the company Deltatec. This way, we also got an insight into the world of work, which was important to us.

In the following, we will first discuss state-of-the-art broadcast concepts to understand the problem. Then, we will study in detail and compare the different decoding solutions implemented during this thesis. Finally, we will establish a retrospective analysis of the work done as well as the perspectives of future work that the thesis opened.

1.1 Introduction to Deltatec

Deltatec is a state-of-the-art technology electronics and **IT** company founded in 1986 and located at Ans in Belgium [50]. This company is mainly specialized in image technologies for a wide range of markets such as aerospace, **machine vision**, and **TV** broadcasting. At its creation, this company was exclusively a hardware design house. And still, today hardware plays a major role in the company since projects in real-time video require advanced management of communication channels, memories, processors, **FPGAs**[1], and **PCBs**[2]. However, with the evolution of technology and the growing complexity of the solutions demanded by customers, Deltatec logically had to master and expand in the software field. This company is constantly growing over the years and aims to adapt to the rapid changes in world of technology, whether in space or even in TV broadcasting. Moreover, Deltatec tends to expand in different fields. As a matter of fact, in the near future, Deltatec intends to expand into the medical sector.

Concerning the collaboration with Deltatec in my Master thesis, the embedded software engineer team welcomed me and provided me with all the necessary development material. Because of the end of health crisis, I worked from home, as well as in the company. This flexibility in the workplace was one of the first challenges I had to solve. This problem and the solution adopted will be discussed at greater length in section 3.1.

1.2 Objectives

As mentioned before, this thesis put into practice very high-performance multimedia decoding for embedded systems and thus follow the increasing requirements of the broadcasting market. To carry out such a study, in agreement with Deltatec and more precisely with my company referent, Julien JEMINE, we had set the objectives of this thesis even before the beginning, the french official document of which is attached in Appendix A. In this section we will go through them, keeping in mind that this list of tasks was voluntarily flexible. That is, throughout the course of the work, meetings were held to review the objectives and reprioritize if necessary. As a result, some of the tasks mentioned in this section were not implemented in this thesis while some other tasks were implemented in this thesis without being initially thought of.

The objective of this thesis was to receive multimedia streams over **IP**, namely **NDI** or NDIHX[3] streams on an NVIDIA Jetson Xavier AGX developer kit, decode them, and transmit them on one or several **HDMI**[4]. In other words, the aim was to provide an efficient and effective decoding solution on the embedded system provided. Consequently, the realization of this work required the mastering of the platform and its associated developing environment. The two main video coding formats used during this project were H.264 and H.265. In order to decode efficiently, the decoding had to use the platform's hardware accelerations. In addition, the HDMI output had to be through a Deltatec **P**eripheral **C**omponent **I**nterconnect **e**xpress (**PCIe**) card manipulated using its `VideoMaster API`. To complete this process, if time allowed it, the objective of this thesis was to implement the reverse process, *i.e.* the reception of HDMI streams, the encoding, and the transmission of network streams over IP.

Another part of this thesis was to implement a decoding solution using `CUDA`. This solution would complement the NVIDIA platform's hardware acceleration-based solution. This second decoding solution would be integrated into a unique decoding application.

[1] **FPGA**, which stands for **F**ield-**P**rogrammable **G**ate **A**rray, is an integrated circuit designed to be (re)programmed.

[2] **PCB**, which stands for **P**rinted **C**ircuit **B**oard, is a board with circuits that connect electronic components together.

[3] See section 2.5 for more details.

[4] **HDMI** stands for **H**igh-**D**efinition **M**ultimedia **I**nterface.

Finally, this project also proposed to add **video overlays** on the output stream. These overlays could either come from another video input, from a **GPU**[5] buffer, or from a **G**raphical **U**ser **I**nterface (**GUI**) technology running on the embedded **Linux**.

Among other products, Deltatec is constantly producing new state-of-the-art video capture cards. Like most high-tech products, one of the challenges facing their product is compatibility. Indeed, the more compatible (yet robust) the product, the larger the list of potential customers. The idea of this thesis is therefore not insignificant. Indeed, being able to efficiently decode quality video on an embedded system is a demand expressed in the market. Therefore, the possibility of extending the compatibility of Deltatec products by offering video acquisition solutions on embedded systems, such as NVIDIA, is a great opportunity.

1.3 Embedded systems

As explained in length in Steve Heath's book [1], an embedded system is a microprocessor-based or microcontroller-based system of hardware and software designed to perform dedicated functions, either as an independent system or as a part of a larger system. Embedded system applications range from digital watches and microwaves to hybrid vehicles and avionics. As much as 98% of all microprocessors manufactured are used in embedded systems. It is important to note that these systems are like mini-computers with their own capabilities.

The first modern, real-time embedded computing system was the Apollo Guidance Computer. This embedded system was designed to collect data automatically and provide mission-critical calculations for the Apollo Command Module and Lunar Module. In 1971, Intel released the first commercially available microprocessor unit. In 1978, the National Engineering Manufacturers Association released a standard for programmable microcontrollers, improving the embedded system design. And by the early 1980s, memory, input, and output system components had been integrated into the same chip as the processor, forming a microcontroller. From then on, microcontroller-based embedded systems have been incorporated into every aspect of consumers' daily lives, from credit card readers and cell phones, to traffic lights and thermostats.

The industry for embedded systems is expected to keep growing rapidly, driven by the continued development of **Artificial Intelligence** (**AI**), **Virtual Reality** (**VR**), and **Virtual Reality** (**AR**), **Internet of Things** (**IoT**), and others. Cognitive embedded systems will be at the heart by providing advantages such as reduced energy consumption, improved security for embedded devices, and visualization tools with real-time data.

1.4 NVIDIA Jetson Xavier AGX developer kit

The NVIDIA Jetson AGX Xavier Developer kit (see Figure 1.1) is an AI computer widely used for autonomous machines, delivering the performance of a GPU workstation in an embedded module [32]. One should understand that this embedded module would then be part of a larger system as the computation unit of, for instance, a robot or a drone.

As well explained in *NVIDIA Jetson AGX Xavier Developer Kit - Introduction* [28], the development kit allows to build and test a solution locally on the module contained in the developer kit to produce this solution on a large scale in Jetson Xavier AGX modules often used in autonomous machines. Nowadays, modern AI requires significant computation power at the edges. The use of autonomous machines continues to develop rapidly with applications, for example, in industry, healthcare, and delivery for example.

[5] **GPU**, which stands for **G**raphics **P**rocessing **U**nit, is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device [52].

Figure 1.1: NVIDIA Jetson Xavier AGX developer kit

Source: `https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit`

1.4.1 Autonomous machine key features

Autonomous machines provide four main features listed here below [32].

1.4.1.1 Bandwidth

Nowadays, autonomous machines typically use multiple high-definition sensors for perception while the pipe back to a data center over the cellular connection is relatively small. Moreover, as mentioned earlier, the world of autonomous machines is growing. The automatic transfer of perceived data at the edge would undoubtedly cause network bottlenecks.

1.4.1.2 Latency

Local platforms that are moving need real-time vision and perception to run correctly and provide safe navigation and path planning.

1.4.1.3 Privacy

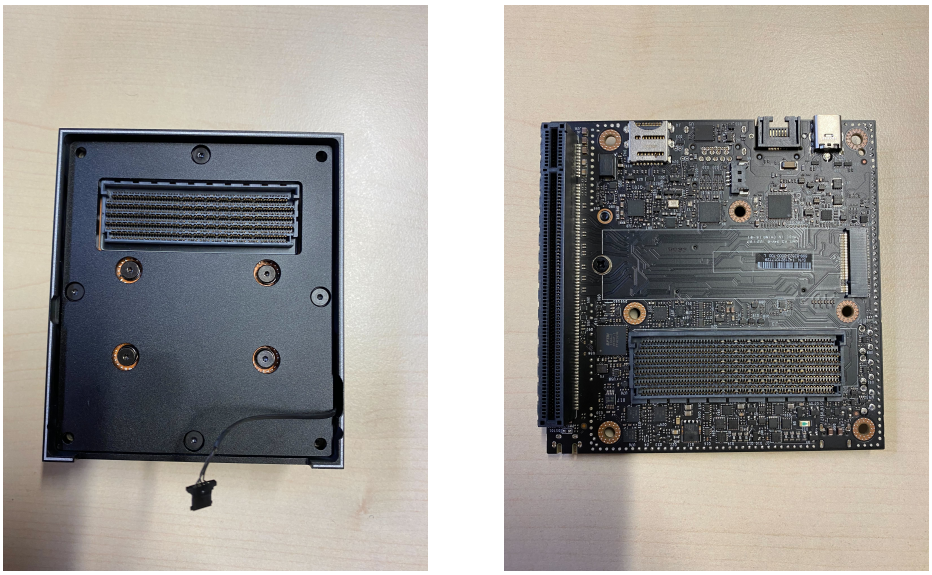
Many of the autonomous machines are operating on the world. As a result, the data collected may be of a sensitive nature or contain personal information. Therefore, the transmission of this information over the air is avoided as much as possible.

1.4.1.4 Availability

Even with the prevalence of 4G **L**ong-**T**erm **E**volution (**LTE**) in urban areas, there are still dead zones or degraded service, more precisely in rugged terrain or inside some large buildings. Consequently, a cellular connection should be relied on for mission-critical applications.

1.4.2 Hardware specification

The NVIDIA Jetson AGX Xavier Developer Kit is nothing but one of the fastest ways to start prototyping with autonomous machines. The two main components of this kit are the Jetson Xavier module itself and the carrier board that it connects to as can be seen in Figure 1.2.



(a) AGX module of the developer kit (b) Carrier board

Figure 1.2: Developer kit components

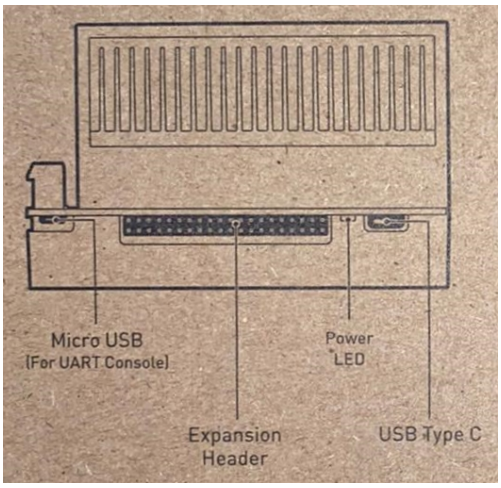
According to the NVIDIA Developer team [28], the Jetson Xavier module is a complete AI computer for autonomous machines. It has the performance of a high-end GPU workstation under 30 Watts. Moreover, the different NVIDIA platform's power modes provide more flexibility to both users and programmers. The details of the main different power modes can be found in Figure 1.3 below.

	MAXN	10 W	15 W	30W_ALL
CPU Cores	8	2	4	8
CPU Max Freq. (MHz)	1377	520	6700	900
GPU Max Freq. (MHz)	2265.6	1200	1200	1200
Memory Max Freq. (MHz)	2133.6	1066	1333	1600

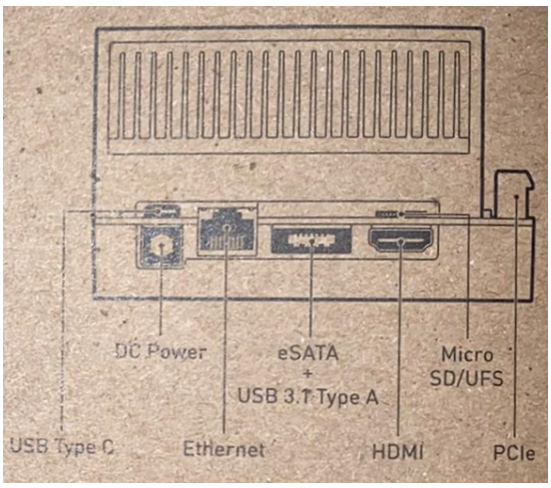
Figure 1.3: Power modes

Source: *Moving Medical Image Analysis to GPU Embedded Systems: Application to Brain Tumor Segmentation* [34]

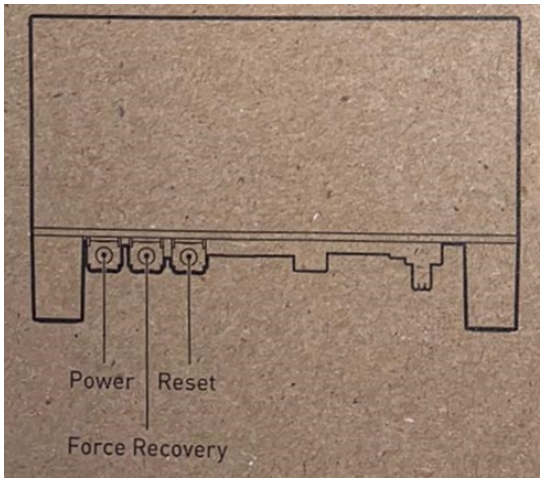
This module is a technological jewel, capable of more than thirty trillion operations per second (30 TFLOPS) for **deep learning** and machine vision tasks. So much power is useless if data cannot be used in the module for processing or signals cannot be got out of this module for interaction with other devices. Therefore, NVIDIA put a great effort into designing a module as usable as possible through the choice of connectors with a 699 pin mirror mezzanine connector that supports all of the high-speed **I**nput/**O**utput (**I**/**O**) of Jetson Xavier. In order to get an idea of the communication performance of the connector, one should note that it can reach up to 56 Gigabits per second of bandwidth. The different connections are shown in the following Figures 1.4 and 1.5.



(a) Front view



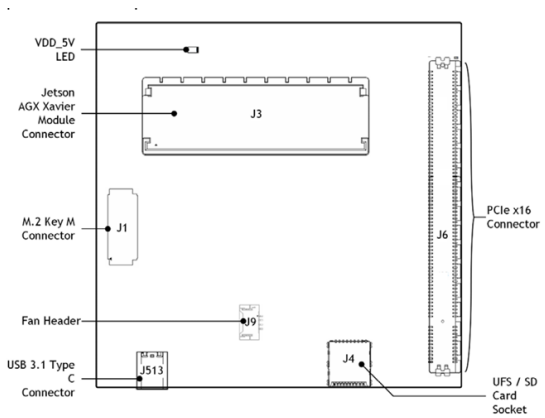
(b) Rear view



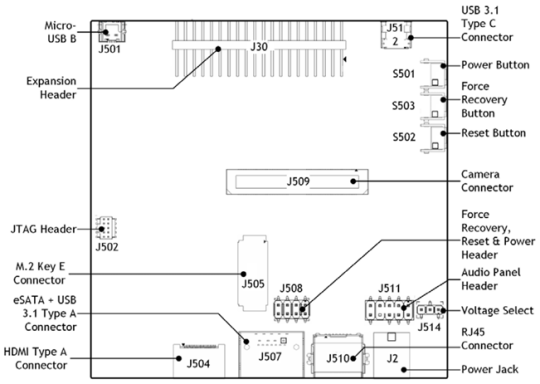
(c) Right view

Figure 1.4: Developer kit views [31]

When it comes to the carrier board (see Figure 1.5), NVIDIA tries to make the developer's life as easy as possible by providing a small board while delivering the maximum possible number of options for connections like mass storage, input, **GPIO** [6] for control and display.



(a) Top view



(b) Bottom view

Figure 1.5: Developer kit carrier board views [31]

[6] A **GPIO**, **G**eneral-**P**urpose **I**nput/**O**utput, port handles both incoming and outgoing digital signals. As an input port, it can be used to communicate to the CPU the ON/OFF signals received from switches, or the digital readings received from sensors.

As mentioned earlier, the AGX developer kit (which contains a specific AGX module) is used to develop and test software in a pre-production environment. On the other hand, traditional AGX Jetson modules are designed for deployment in a production environment throughout their operating lifetime. One should note that the only difference between the native AGX module of the developer kit and an independent one is that the native one (see Figure 1.2a) is provided with a specific thermal solution created for the developer kit whereas the basic Jetson Xavier module (see Figure 1.6) comes with integrated thermal transfer plate to simplify the integration with a customized thermal solution. The master piece of this thesis was the developer kit.

Figure 1.6: NVIDIA Jetson Xavier AGX module

Source: `https://www.nvidia.com/fr-fr/autonomous-machines/embedded-systems/jetson-agx-xavier/`

Chapter 2

State-of-the-art

In order to achieve the objectives of this thesis, it was necessary to understand and master a large number of important theoretical concepts that we will discuss in detail in this chapter. Indeed, world of multimedia broadcasting is a mature world full of complex concepts allowing us to reach the current performances and even improve them. In this chapter, we will review the main principles of the video world by studying mainly video formats and their coding techniques, which have given rise to well-defined video coding standards, without forgetting to discuss the revolution provided by video over **I**nternet **P**rotocol (**IP**), which has considerably reduced the costs associated with streaming production.

2.1 Video streaming

As further explained in Huifang Sun, Anthony Vetro, and Jun Xin's article [7], video streaming addresses the problem of transferring video data as a continuous stream. This data becomes bigger and bigger as the video quality keeps increasing nowadays. With streaming, the end-user can start displaying video or multimedia data before the entire file has been transmitted. To achieve this, the bandwidth efficiency and flexibility between video servers and equipment of end-users are particularly important, which leads to challenging problems. A typical video streaming system is shown in the following Figure 2.1.
This section is also inspired by David R. Bull and Fan Zhang's book [35].

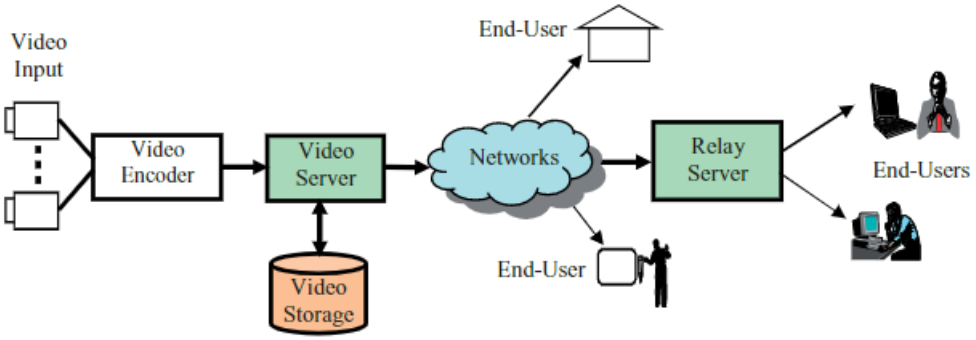


Figure 2.1: Video streaming system [7]

Streaming is used all over the world and most of the time by millions of users at the same time. Moreover, nowadays, the resolution of a video, directly related to its quality, is generally at least 1080p[1], while 4K (four times heavier than 1080p in terms of amount of data) is becoming a quality standard on a large scale. Besides, 8K resolution, which is 16 times heavier than a 1080p video, is very soon the next quality standard in broadcasting world.

2.1.1 Unfeasibility of raw video data

As a numerical example, if one considers a system that uses 24 bits to model a color, as most multimedia systems do, a 1080p resolution frame represents about 5.93 MB ($1080 \times 1920 \times 24$ bits)[2]. This means that in terms of raw data quantity, a 1080p video of only one minute with the standard frame rate of 25 frames per second[3] represents 8.69 GB ($5.93 \frac{\text{MB}}{\text{frame}} \times 25 \frac{\text{frame}}{\text{sec}} \times 60\,\text{sec}$). This raises the question of network congestion. Indeed, network resources are limited, including router processing time and link throughput. Resource contention may occur in networks in numerous circumstances. As a matter of fact, network congestion typically arises when a network node or link carries more data than it can handle. In such a situation, the quality of the delivered services inevitably reduces, and typical effects including queueing delay, packet loss, or the blocking of new connections occur.
On the other hand, if one would even assume an internet network that cannot be congested, one must look at the time required to transfer such files. In Belgium, the best internet connections for a private individual (via fiber optics) can approximately reach up to 1 Gbps (Gigabyte per second) in download and 100 Mbps (Megabyte per second) in upload [58]. Therefore, it would

[1] 1080p often refers to 1920 pixels wide and 1080 pixels high. However, the notations 1080p, 720p, 480p, *etc.* mainly specify the height of the image and do not really constrain the width of the image, although there are standards. Thus, 1080p is $width \times 1080$ resolution, 720p is $width \times 720$ resolution and so on, where *width* can be anything.

[2] Using the International System of Units (SI) [68]: 1 Gigabyte (GB) = 1024 Megabytes, 1 Megabyte (MB) = 1024 Kilobytes, 1 Kilobyte (KB) = 1024 Bytes, and 1 Byte = 8 bits.

[3] **PAL**, which stands for **P**hase **A**lternate **L**ine, is the video format standard mainly used in Europe and its frame rate is 25 fps.

take about 9 seconds to download this small video and up to about 12 minutes to upload.
This small example allows us to realize that in order for streaming to be usable, it was necessary to find a way to drastically compress data video. Nowadays, the file size for a one-hour 1080p video is only 1.2-1.4 GB.

There exists a variety of solutions to achieve such a data compression. The two main ones are scalable and non-scalable video coding solutions [7] whose compression techniques will be discussed in section 2.3. The first one is a much older solution while the other is the kind of solution used nowadays.

2.1.2 Non-scalable video coding

In non-scalable video coding, video content is encoded independent of the actual channel properties. Therefore, in this kind of coding, it is difficult to adaptively stream non-scalable video contents to heterogeneous client terminals over time-varying communication channels. As a result, since the encoded data may not be supported over a channel, one must switch among multiple pre-encoded non-scalable bitstreams. Indeed, for video bitstreams distribution, the video and relay servers are generally responsible for matching the output data to the available channel resources and ultimately the client's device capabilities. For non-scalable video data, the server may transcode the bitstream to reduce the bit rate, frame rate, or spatial resolution. Alternatively, it may select the most appropriate bitstream from multiple pre-encoded streams having different quality, spatial resolution, and others.
Non-scalable video coding focuses on coding efficiency, and the compression is optimized at a pre-specified rate.

2.1.3 Scalable video coding

With **S**calable **V**ideo **C**oding (**SVC**), video needs to be encoded only once. Then by simply truncating layers or bits from the single video stream, lower qualities and spatial and temporal resolutions could be obtained. Of course, this scalability does not go without a cost as, in practice, all scalable video coders suffer a loss in compression efficiency relative to state-of-the-art non-scalable coders.
The most widely spread solution is H.264 coding format which is a SVC standard.

More generally, the compression used in multimedia is said to be lossy compression, which means that compression is irreversible, unlike lossless compression. In other words, it is impossible to build a decoder that can restore exactly the original information from the encoded data via lossy compression.

2.2 Video Format

A video is nothing more than a sequence of images called **frames**. When we talk about the format of a video, we are really talking about the format of each frame. In multimedia world, there are many different formats each implementing its own color model. A color model is an abstract mathematical model describing how colors can be represented as tuples of numbers, typically as three or four values of color components. When this model is associated with a precise description of how components are to be interpreted (viewing conditions, *etc.*), the resulting set of colors is called color space. In this section we will describe, using the literature on this subject [38, 65, 4] common color spaces resulting in specific video formats, namely RGB, YUV, and NV.

2.2.1 RGB(A)

Without a doubt, the format most known by the general public given the simplicity of the concept behind it, the **RGB** format (**R**ed **G**reen **B**lue) is characterized by 3 data planes: red plane, green plane, and blue plane as illustrated in the following Figure 2.2. The principle of the RGB format is to compose any color from 3 primary colors that are red, green, and blue. Generally, the intensity of each of primary color is specified with 8 bits which explains the 24 bits used to describe a single **pixel** in an image stated earlier in section 2.1. The A, which stands Alpha, in **RGBA** is the data plane corresponding to the opacity and is, for instance, used in the png format of a photo.

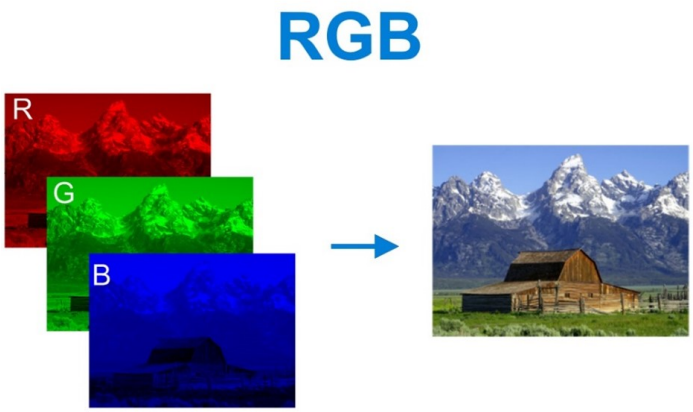


Figure 2.2: Picture decomposition in data planes [76]

2.2.1.1 Computer image

The notions of frame, pixel, and even resolution have already been mentioned above. But before going into further details, it would be wise to recall the structure of a computer image based on *Raster images in computing* chapter of Kaufmann's book [18]. A digital image, S, is an image composed of a set of picture elements, also known as pixels or samples, s, with spatial dimensions (often called resolution) $X \times Y$ as shown in Figure 2.3.

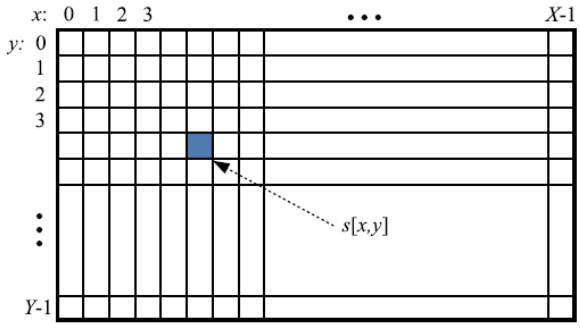


Figure 2.3: Image sample array [18]

Each sample, $s[x, y]$, has a finite, discrete quantities of numeric representation for its intensity or gray level that is an output from its two-dimensional functions fed as input by its spatial coordinates denoted with x, y on the x-axis and y-axis, respectively. A pixel, which is a sample of an original image, is the smallest controllable element of a picture represented on the screen. One should note that obviously, the more the samples, the more accurate the representations of the original image, thus the greater the resolution.

In the case of the RGB format, each element of S is itself a vector s with three dimensions representing the intensities of the red, green, and blue components respectively.
The matrix form is as follows:

$$\mathrm{S} = \begin{bmatrix} \mathrm{s}[0,0] & \mathrm{s}[0,1] & \dots & \mathrm{s}[0,X-1] \\ \mathrm{s}[1,0] & \mathrm{s}[1,1] & \dots & \mathrm{s}[1,X-1] \\ \vdots & \vdots & & \vdots \\ \mathrm{s}[Y-1,0] & \mathrm{s}[Y-1,1] & \dots & \mathrm{s}[Y-1,X-1] \end{bmatrix}$$

where

$$\mathrm{s}[x,y] = [s_R[x,y] \quad s_G[x,y] \quad s_B[x,y]]$$

The intensity of each pixel is thus a combination of three primary colors. The following Figure 2.4 depicts the composition of an image in pixels.

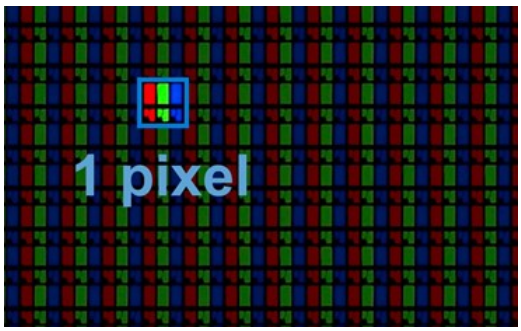


Figure 2.4: RGB pixel [76]

Contrary to what one might have thought, although the RGB signal is simple in concept and is the only internal signal understandable by a conventional **L**iquid **C**rystal **D**isplay (**LCD**) television, the RGB format is not the one chosen for most of the signals circulating in cables and in the air. The explanation behind this choice is detailed in the next section 2.2.2.

2.2.1.2 Color space

This subsection is inspired by [33].
While in the physical world as we know it, colors are wavelengths of light, it was necessary to define a relationship between these wavelengths and the red, green, and blue values used in video formats. This correspondence was made possible by the use of so-called color spaces.

2.2.1.2.1 CIE 1931 chromaticity diagram In 1931, the **I**nternational **C**ommission on **I**llumination (**CIE**) established a means of representing colors visible to humans using the CIE RGB space (see Figure 2.5).

This diagram represents the complete range of colors visible to the average human eye. Using wavelength markers along the edges, one can express every color on the visible spectrum as a set of coordinates in diagram 2.5.

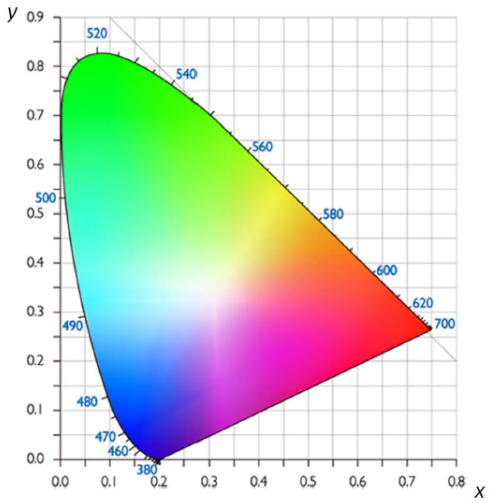


Figure 2.5: Chromaticity

Source: `https://fr.wiktionary.org/wiki/chromaticit%C3%A9`

2.2.1.2.2 Wavelength coordinates to RGB values To transform these coordinates into the RGB components as described above (subsection 2.2.1.1), one must define a four-point triangle (red, green, blue, and white points) surrounding the chromaticity diagram (see Figure 2.6)[4].

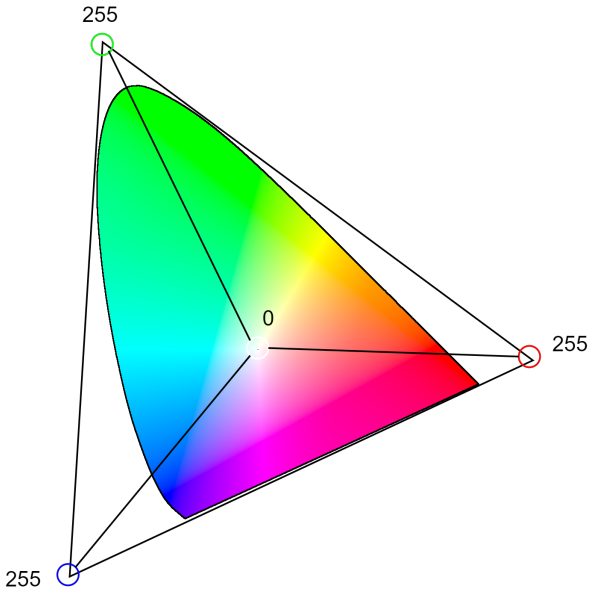


Figure 2.6: Four-point chromaticity triangle [33]

These points describe where the color component values lie on the chromaticity diagram. However, one should know that those RGB values are only meaningful relative to the considered color space. Indeed, if one defines the four points of the triangle in diagram 2.6 in different locations, then the same physical color will be represented by a different set of RGB values. Hence, in order to obtain a consistent color model, one must know both the RGB values and the color space coordinate system corresponding to these values.

[4]One assumes that each color channel is encoded on 8 bits, which means that there are 256 (2^8) shades for each color component.

2.2.1.2.3 Color space standards One should note that in the visible color palette (Figure 2.5), the human eye is able to differentiate between about ten million different shades, which is called color depth. However, calibrating cameras and monitors to reliably reproduce each of those colors is expensive and wasteful in practice. Indeed, some colors, such as extremely saturated colors, are not perceived on a daily basis. Therefore, it is possible to save some resources by calibrating devices to work in smaller subsets of the visible spectrum. Consequently, certain standards of color spaces are born, some of them are illustrated in Figure 2.7.

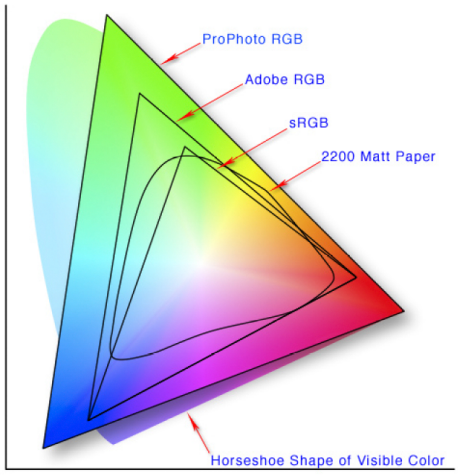


Figure 2.7: Examples of color space standard

Source: `https://en.wikipedia.org/wiki/Color_space`

The four-point triangle of these standards is thus smaller but cheaper to work with. As there exist multiple standards, one must worry about color space conversion, which consists in moving from one space to another without affecting the color one wants to represent.

2.2.1.3 sRGB

As just explained, the color model itself does not define absolute colors. Indeed, RGB values are relative to the considered color space. Therefore, the RGB color model itself does not define what is meant by red, green, and blue colorimetrically. Hence, the results of mixing them are not specified as absolute, but relative to the primary colors. When the exact chromaticities of the red, green, and blue primaries are defined, the color model becomes an absolute color space. An example of such a color space is **sRGB**, where s stands for standard. Absolute color space is said to be device-independent color space. This is why sRGB color space is used as default in most modern displays unless another color space is specified.

2.2.2 YUV - YCbCr

The audiovisual world has not always been in color. Indeed, the first televisions broadcast images in black and white through a signal called luminance and noted Y. Therefore, the luminance represents the brightness of an image. At the time of the creation of the first television sets in color, one could have thought that the historical signal, luminance was going to be abandoned for a signal of type RGB. However, to ensure backward compatibility with monochrome televisions, via a single signal, engineers of the time developed the YUV format. YUV preserves the historical signal that is the luminance and superimposes the color information called chrominance, noted UV, onto the luminance signal. From then on, black and white TVs only used luminance while color TVs used both luminance and chrominance. Indeed, the YUV signal is designed in such a way that chroma does not significantly interfere with the luma signal. This allowed monochrome televisions to exploit the combined signal as a grayscale image.

Historically, the YUV (more precisely Y'UV) refers to an analog signal (PAL or **SECAM** TVs, or monitors) while the term YCbCr [74] (Y'CbCr or YPbPr/Y'PbPr) is used for digital signals (TVs/**HD** monitors) even if they both refer to the same division into three planes [75]:

- Y = gray level (luminance)
- U / Cb = Y - Blue, which gives a color between blue and green.
- V / Cr = Y - Red, which gives a color between yellow and red.

Beyond the backward compatibility with monochrome TV sets, YUV is able to be compressed by almost half and thus reduce the bandwidth, without even the slightest difference being perceptible to the naked eye. It is this last reason which pushes the audiovisual world to continue to work with this format until today when all the television sets are polychrome. This efficient compression is achievable thanks to the concept of chroma subsampling which will be further discussed in subsection 2.3.1.1.

Therefore, in practice, the RGB images captured by cameras are transformed into YUV/YCbCr format, lighter than the RGB format, to be then transmitted (antenna, satellite, *etc.*) to a TV set that always ends up by retransforming them in turn to the RGB format to display them. Conversion from RGB format to YUV format is also called a color model conversion. The cycle followed by a multimedia stream is shown in Figure 2.8.

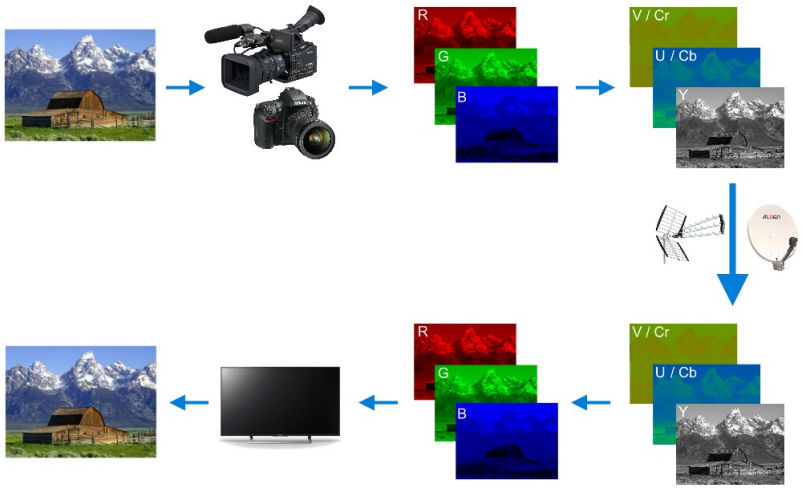


Figure 2.8: Video stream path [76]

In Europe, TV transmission relies on a PAL system. In this kind of system, the color model conversion is given by the following expression.

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & -0.436 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

Where each component R'G'B' is the gamma corrected[5] version of the RGB format.

[5]Gamma correction controls the overall brightness of an image. Therefore, reproduce colors accurately also requires some knowledge of gamma.

2.2.3 NV

As for the group of NV formats (NV12, NV16, and NV24), it is a format quite close to the YUV format described previously but less known by the general public. We thought it might be wise to talk about it since this format was used in the decoding application. Like YUV format, NV format is based on luminance and chrominance information. However, it differs from the YUV format simply in the arrangement of the data. Indeed, as opposed to the YUV format which has three data planes, the NV format contains only two planes: one for the luminance and another for the chrominance. In this last plane, Cr and Cb are interlaced [21].

2.3 Coding techniques

As mentioned earlier, video compression is crucial for video streaming to be usable. It consists of two main parts: spatial coding and temporal coding which will be extensively described in this section using notably David Austerberry's book [5].

2.3.1 Spatial (intra-frame) coding

As a reminder, a video is composed of a series of frames, each frame being an image. Spatial coding is nothing more than classical image compression applied to each video frame. Image compression consists mainly of two principles, namely chroma subsampling and **D**iscrete **C**osine **T**ransform (**DCT**). In this section, one will describe in detail what these two principles are.

2.3.1.1 Chroma subsampling

As clearly explained in *Introduction to Luma and Chroma* [12], Chroma subsampling consists of reducing the resolution of the Cb and Cr (or U and V) color components. As mentioned previously, this resolution reduction for the color components is, in most cases, provided that a suitable subsampling format is chosen, imperceptible to the naked eye since the human eye is much more sensitive to luminance (Y) than to the color components (Cb and Cr) [37]. The RGB format, on the other hand, does not allow for an easy separation between light intensity information and color information. Consequently, this is the main reason why video storage and transmission are done in YUV format rather than RGB format.

The sampling structure is defined from three numbers on a matrix of 8 pixels (4 x 2). The first number is the number of luminance samples (Y) per line, the second number is the number of chrominance samples (Cb / Cr) on the first line of pixels, and the third number is the number of chrominance samples (Cb / Cr) on the second line of pixels. In order to better understand what this refers to, we will image the three most used subsampling for the YUV format in the following.

2.3.1.1.1 4:4:4 The 4:4:4 format corresponds to a raw one, without compression per frame[6]. There is thus no quality loss in this type of format. Each final image pixel is generated from a luminance pixel, a Cb chrominance pixel, and a Cr chrominance pixel. It is important to note that, in this configuration, there is no difference between a YUV format signal and an RGB format signal in terms of bandwidth usage. Figure 2.9 depicts the 4:4:4 chroma subsampling.

[6]As chroma subsampling is not the unique compression technique, this does not mean that the video will not be compressed at all, using 4:4:4 subsampling, for instance, see subsections 2.3.1.2 and 2.3.2.

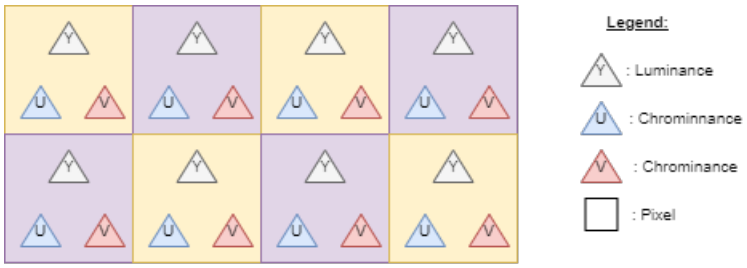


Figure 2.9: 4:4:4 subsampling

It is not surprising that this format is not widely used, since a very high bandwidth represents a significant cost. As a result, 4:4:4 subsampling is only used for applications where higher quality is required. This is, for example, the case in the computer world (computers, video games production) but also in the professional cinema world where the objective is to improve images with the best possible quality beforehand to embed some special effects.

2.3.1.1.2 4:2:2 With the 4:2:2 subsampling, the horizontal resolution of the chrominance is divided by two. In other words, it is the same color Cb that will be used for the final rendering of two pixels. The same applies to the Cr color, as illustrated in Figure 2.10. As a result, there are 4 Cb- and Cr-values for 8 pixels while there are 8 Y-values for as many pixels.

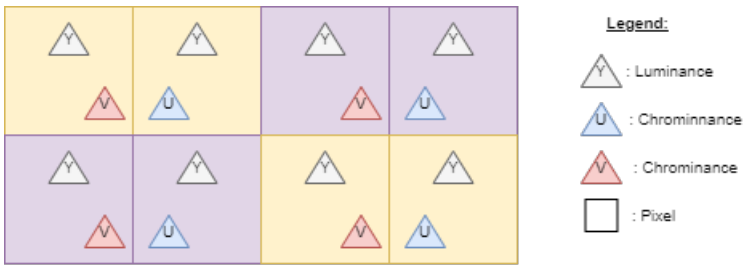


Figure 2.10: 4:2:2 subsampling

With a bitrate reduced by 33% and a difference mostly invisible to the naked eye, this format is also intended for professionals.

2.3.1.1.3 4:2:0 Finally, the 4:2:0 format is the subsampling used for the general public, *i.e.*, for the world of television, Blu-Ray, video games, and others. This time, color images (Cb and Cr) are halved in horizontal and vertical resolutions, as in the following Figure 2.11. This means that most of the time when we are confronted with video content, only one pixel out of four in the image contains original color information, while the three others contain color information that is repeated from original one.

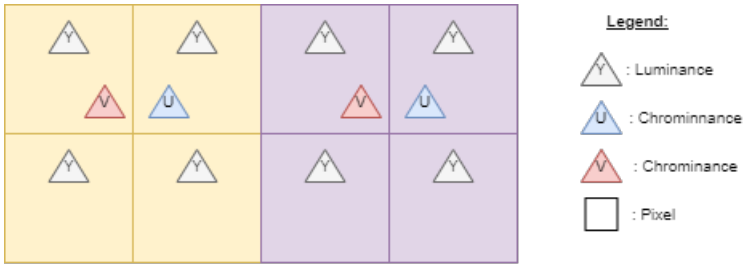


Figure 2.11: 4:2:0 subsampling

The direct consequence of such an information reduction is that the bandwidth is reduced by half compared to the 4:4:4 format. Again, the difference in image quality is barely noticeable

in the vast majority of cases due to the greater sensitivity of the human eye to light than to color.

One should note that there are many other subsampling formats such as 4:2:1, 4:1:1, or 5:0. However, these formats are much less common.

2.3.1.2 Discrete Cosine Transform

This subsection is inspired from scientific article [8] and literature [14, 39].

2.3.1.2.1 Frequency-dependent contrast sensitivity In addition to the fact that the human eye is more sensitive to light intensity than to color, which was exploited for image compression through chroma subsampling, there exists another characteristic of the human eye that one can take advantage of for image compression, namely the frequency-dependent contrast sensitivity. Indeed, this sensitivity translates into the obvious fact that it is easier for the human eye to miss small objects or fine details in a picture compared to the large ones [37]. Figure 2.12 illustrates this phenomenon.

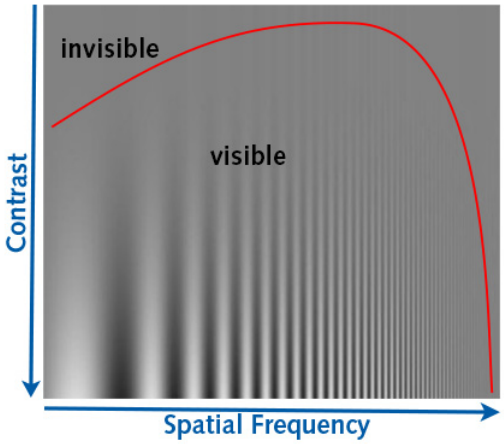


Figure 2.12: Frequency-dependent contrast sensitivity

Source: `https://www.psychophysics.uk/spatial-contrast-sensitivity/`

2.3.1.2.2 Block division Even if this sensitivity slightly varies from person to person, one can notice that the details in the visible region are more visible than the ones in the invisible region, since the human visual system is more sensitive to brightness variations in the range of spatial frequencies of the visible region. Therefore, as the details from the right top corner of Figure 2.12 are barely visible, image compression tends to save some data representing almost unnoticeable details. Spatial coding reaches this objective by dividing the image into (8×8) blocks and quantizing them in a frequency-domain representation. This is done by comparing each one of these (8×8) blocks with 64 frequency patterns. The 64 frequency patterns are illustrated on the right of Figure 2.13, where the spatial frequency increases from left to right and top to bottom.

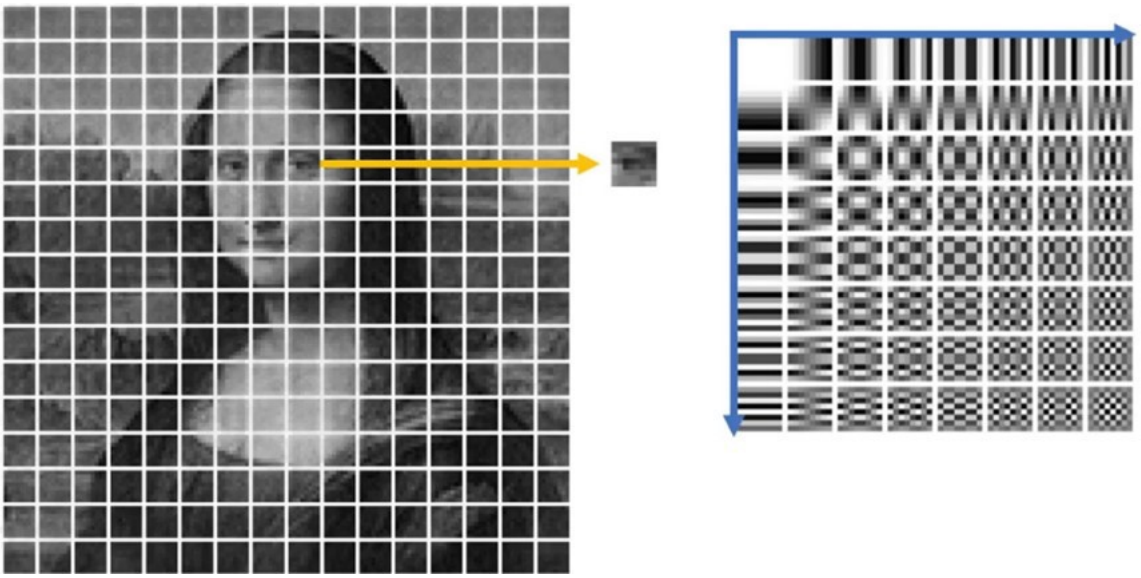


Figure 2.13: Image block division and quantization [29]

This process decomposes the image into its frequency components by converting an (8×8) block where each cell represents a brightness level into another one where each block cell represents the presence of a particular frequency component (see Figure 2.14). This method is called the discrete cosine transform.

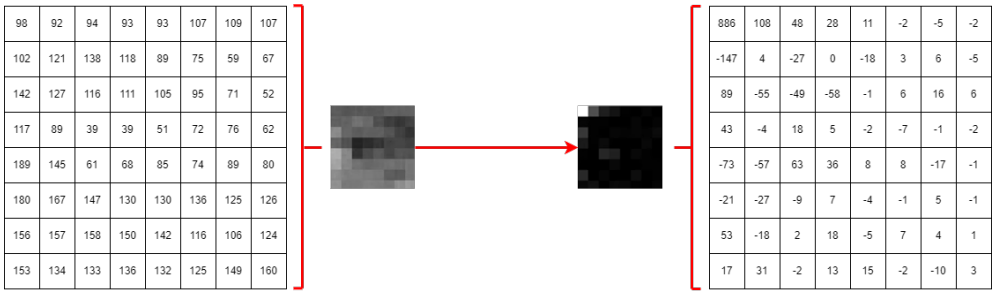


Figure 2.14: Discrete cosine transform [29]

2.3.1.2.3 Mathematical formulas Although there are several transforms of the same kind, such as **K**arhunen–**L**oeve **T**ransform (**KLT**) or **D**iscrete **F**ourier **T**ransform (**DFT**), the discrete cosine transform has proven to be the most efficient trade-off between computational time and energy compaction in practice.
Mathematically, the 8×8 **F**orward DCT (**FDCT**) takes an 8×8 array of 64 sample values (denoted $\boldsymbol{f}$, whose elements are $f_{i,j}$) and produces an 8×8 one of 64 transform coefficients (denoted $\boldsymbol{F}$, whose elements are $F_{u,v}$). The FDCT is expressed by this equation, whose details about the derivations can be found in *Transforms for image and video coding* [39]:

$$F_{u,v} = \frac{1}{4} C(u)C(v) \sum_{i=0}^{7} \sum_{j=0}^{7} f_{i,j} \cos\left[\frac{(2i+1)u\pi}{16}\right] \cos\left[\frac{(2j+1)v\pi}{16}\right];$$

$$C(w) = \begin{cases} \frac{1}{\sqrt{2}}; & w = 0 \\ 1; & w = 1, 2, \ldots, 7 \end{cases}$$

When it comes to the inverse discrete cosine transform, IDCT or DCT^{-1}, it respects the following expression:

$$f_{i,j} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u)C(v)F_{u,v} \cos\left[\frac{(2i+1)u\pi}{16}\right] \cos\left[\frac{(2j+1)v\pi}{16}\right]$$

As the forward and inverse transform involve nearly identical arithmetic, the coding and decoding complexities are similar. Indeed, the DCT is its own inverse (within a scale factor). Consequently, performing the DCT on the transform coefficients would perfectly reconstruct the original samples, subject only to the roundoff error in the DCT and IDCT.

2.3.1.2.4 Quantization In the representation on the right of Figure 2.14, one can easily compress the frequencies, that are less visible by the human eye. Indeed, this is achieved by dividing these frequency components by some constants and then quantizing them as depicted in Figure 2.15.

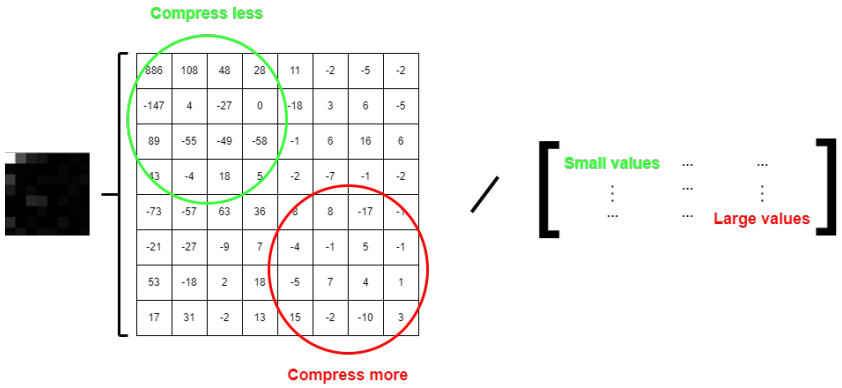


Figure 2.15: Quantization [29]

Consequently, the frequency components that human eyes are less sensitive to are divided by larger constants as compared to the ones that human eyes are more sensitive to. In this context, quantization simply means rounding the result to the nearest integers. Using larger divisors lead to more numbers rounded to zero, hence resulting in higher compression rates but also in lower image qualities. After quantization, the matrix block representation ends up with lots of zeros in the high frequencies, as illustrated on the left of Figure 2.16.

2.3.1.2.5 Zig-Zag arrangement As explained in *Lossless compression methods* [40], such a **sparse matrix**[7] can be stored more efficiently by rearranging the elements. Indeed, if the coefficients are reordered in Zig-Zag, as in Figure 2.16, from top left to bottom right, one can group the zeros.

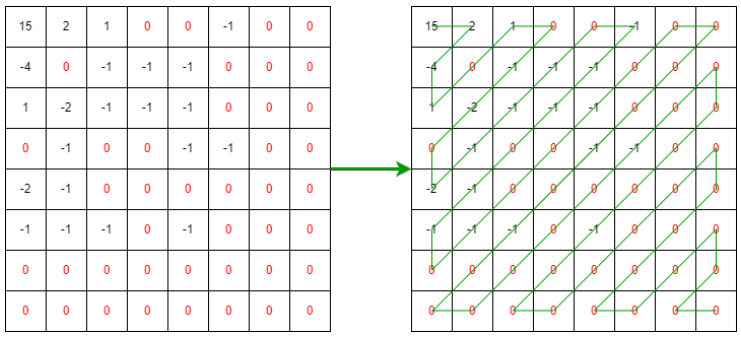


Figure 2.16: Zig-Zag arrangement [29]

From the figure above, the coefficients rearranged in the Zig-Zag order from top left to bottom right gives the following sequence of numbers:

$[15,\ 2,\ -4,\ 1,\ 0,\ 1,\ 0,\ -1,\ -2,\ 0,\ -2,\ -1\,(3\text{x}),\ 0,\ -1\,(3\text{x}),\ 0,\ -1\,(2\text{x}),\ 0,\ -1,\ 0\,(2\text{x}),\ -1,\ 0\,(5\text{x}),\ -1,\ 0,\ -1,\ 0\,(6\text{x}),\ -1,\ 0\,(5\text{x}),\ -1,\ 0\,(17\text{x})]$

[7] A sparse matrix is a matrix containing almost exclusively zeros.

2.3.1.2.6 Run-length encoding Once the zeros of the quantized DCT matrix are grouped, instead of saving each coefficient separately, one can save their value and the number of times they consecutively occur in tuples. For example, in the example illustrated above, the last seventeen 0's are represented as the following tuple: `repeat(0, 17)`. This technique is called run-length encoding and is also used in many other algorithms.

2.3.1.2.7 Huffman coding As the last compression step, image compression applies Huffman coding to further reduce data size. This algorithm belongs to the family of **entropy coding**[8] and consists in encoding the more frequent values with fewer bits and the less frequent values with more bits. As a result, the average number of bits per symbol is reduced.

Both Run-length encoding and Huffman coding are lossless compression methods, meaning that no information is thrown out in these steps. Indeed, compression is achieved solely by storing the data more efficiently.

2.3.1.3 Intra-prediction

A common practice heavily used in inter-frame coding (see subsection 2.3.2 for more details), that can also be exploited in intra-frame coding, is to divide each frame into macroblocks. Each macroblock can be split further into coding units. These can be partitioned even further and cut up in different ways into prediction units, as shown in Figure 2.17.

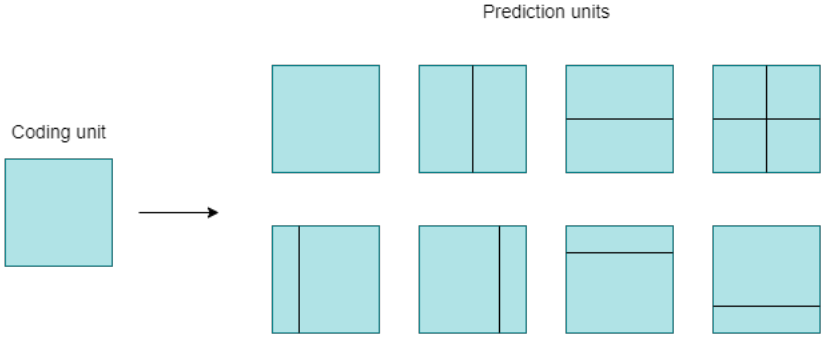


Figure 2.17: Prediction unit division [23]

This division is made to have the possibility to mathematically generate pixel values in a block instead of storing them and, therefore, massively reduce the size of each frame. Indeed, it is less expensive, in terms of amount of data, to store a mathematical function and the frame region to which it applies than the value of each pixel in this same region. This method is called intra-frame prediction and is fully explained in *The block-based hybrid video codec* [42] and *A Block-Matching Based Intra Frame Prediction for H.264/AVC* [6].

Let us consider the example where one has a prediction unit of 4×4 pixels (gray part in Figure 2.18) to understand how it works in practice. Let A (I, J, K, L, and M squares in Figure 2.18) and B (A, B, C, D, E, F, and G squared in Figure 2.18) be two blocks that surround the prediction unit. One can apply various intra-prediction modes on this structure.

[8] Entropy coding is a lossless data compression scheme that is independent of the specific characteristics of the medium.

2.3.1.3.1 DC DC prediction mode consists in filling prediction unit with the average of surrounding pixels.

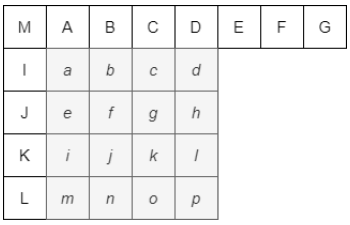


Figure 2.18: Prediction unit example [23]

In the above example (Figure 2.18), DC prediction mode implies that:

$$\{a, b, \cdots, p\} = \left\lfloor \left(\frac{A + B + C + D + I + J + K + L}{8} \right) + 0.5 \right\rfloor$$

This mode is usually used on an outdoor scene with a blue sky where there is no actual pattern to it. It is just a single color.

2.3.1.3.2 Angular function An angular function is a type of mathematical function that consists in filling the pixels of the prediction unit according to a certain direction. This mode is useful when there is a directional correlation such as a line. The following Figure 2.19 illustrates different angular functions suited for the pattern illustrated on the right.

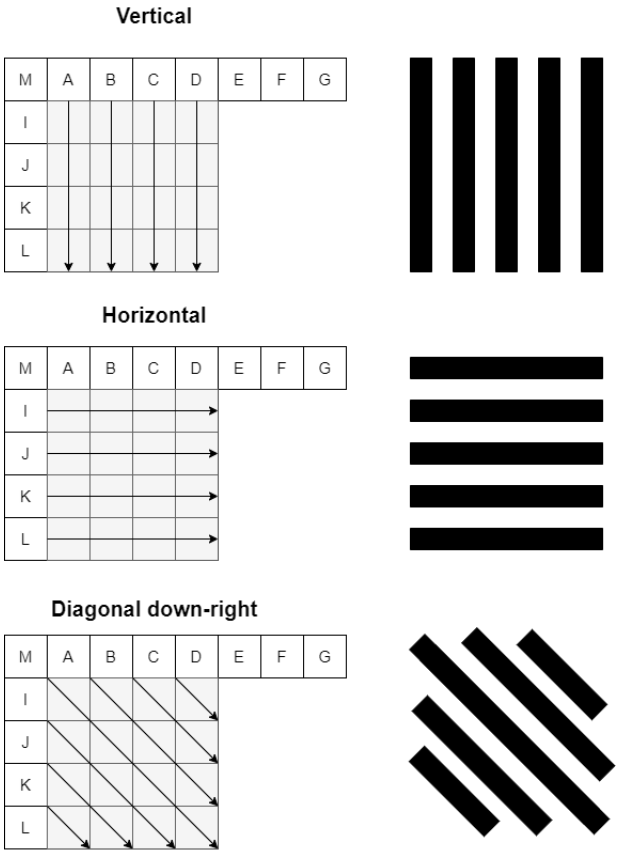


Figure 2.19: Intra prediction angular functions

In these cases, the value of the neighboring pixels is repeated along the direction of the arrow.

2.3.1.3.3 Intra-frame coding flow The following diagram 2.20 summarizes the process of intra-frame coding where green blocks are lossless compression steps and orange ones are lossy compression steps.

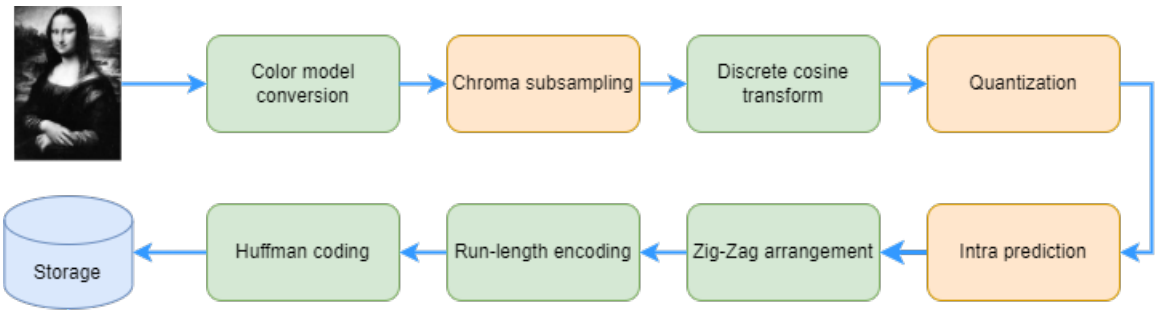


Figure 2.20: Intra-frame coding [29]

When decoding an image, all these steps are reversed. Since some information is lost during the subsampling, quantization and prediction steps, the decoded image will not be identical to the original one. However, the compressed images should almost look like the original ones when a reasonable compression rate is used. Compression artifacts arise when too high compression rates are used. Moreover, these artifacts become more visible as the compression rate increases.

2.3.1.3.4 Directional transform *This paragraph is inspired from An overview of directional transforms in image coding [9].*
Most of the time, the correlations in an image are directional. For example, an image may have a stronger vertical correlation than in all other directions, as in the following Figure 2.21.

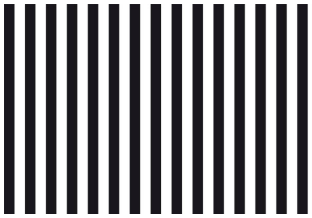


Figure 2.21: Vertically correlated image

Since the human eye is highly sensitive to directional correlations, efficient representation of directional information is extremely important for high-performance image coding. However, the traditional 2D DCT is not very efficient for directional correlations. Indeed, the classical version is implemented separately along with the vertical and horizontal directions, respectively. In practice, such a separable fashion significantly reduces the computational complexity compared to a non-separable 2D transform. However, this separation method effectively encodes only vertically or horizontally correlated images as the basis functions of a separable 2D transform do not support directionalities other than vertical or horizontal. Therefore, the solution to this problem is to apply the **M**ode-**D**ependent **D**irectional **T**ransform (**MDDT**) algorithm to also take advantage of correlations other than horizontal or vertical.

2.3.2 Temporal (inter-frame) coding

Beyond compressing a video data frame by frame, which is called spatial or intra-frame coding and significantly reduces the file size, video compression aims to take advantage of the temporal redundancy between frames in a video, which leads to an even greater compression rate. This process is called temporal or inter-frame coding. This subsection, which is based on *Coding moving pictures: motion prediction* [41], aims to describe the main ideas behind temporal coding. In a typical video, many consecutive frames tend to be nearly identical.

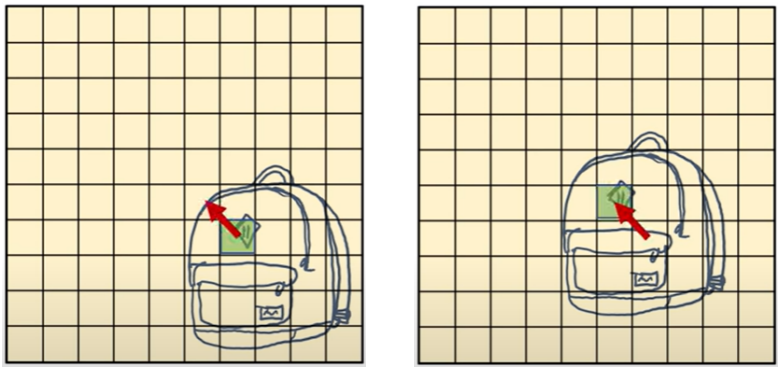
2.3.2.1 Block motion estimation and compensation

Block motion estimation and compensation both work on frames that are divided into blocks called macroblocks. Although it is similar to the block division used in the discrete cosine transform, one should not mix the two kinds of blocks. Indeed, typical macroblock size is (16×16), against (8×8) for a DCT block. But macroblock size can be of varying sizes. Each macroblock can be split further into blocks called coding units, which can go down to 8×8 pixels.

2.3.2.1.1 Stationnary video Let us first consider a video where nothing changes. Hence, all the frames are identical. In such a case, instead of storing each one of these same frames, the encoder can simply keep one frame copy and repeat it as many times as needed, which would lead to lots of space saved.

2.3.2.1.2 Partially changing video In more realistic videos, there are still some parts of the video that do not change from frame to frame. Therefore, one can apply the same idea as the one in the stationary video, but more locally. This can be achieved by comparing each macroblock individually.

2.3.2.1.3 Fully changing video In most of the videos, all macroblocks change between consecutive frames. However, most of these changes are minor. Therefore, instead of checking whether a macroblock has changed or not, a more clever way to compress the data is to search for a given macroblock in the next frame within a neighborhood, as in Figure 2.22. This process is called block motion estimation.



(a) Current frame (b) Next frame

Figure 2.22: Block motion estimation [30]

This process is combined with the process of motion compensation. The latter consists in saving a reference frame and the motion vectors for the blocks. The motion vectors specify in which direction the blocks should move to closely match the next frames. This process is shown in Figure 2.23 here below.

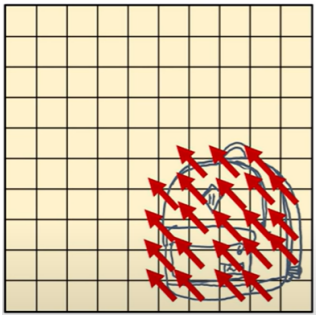
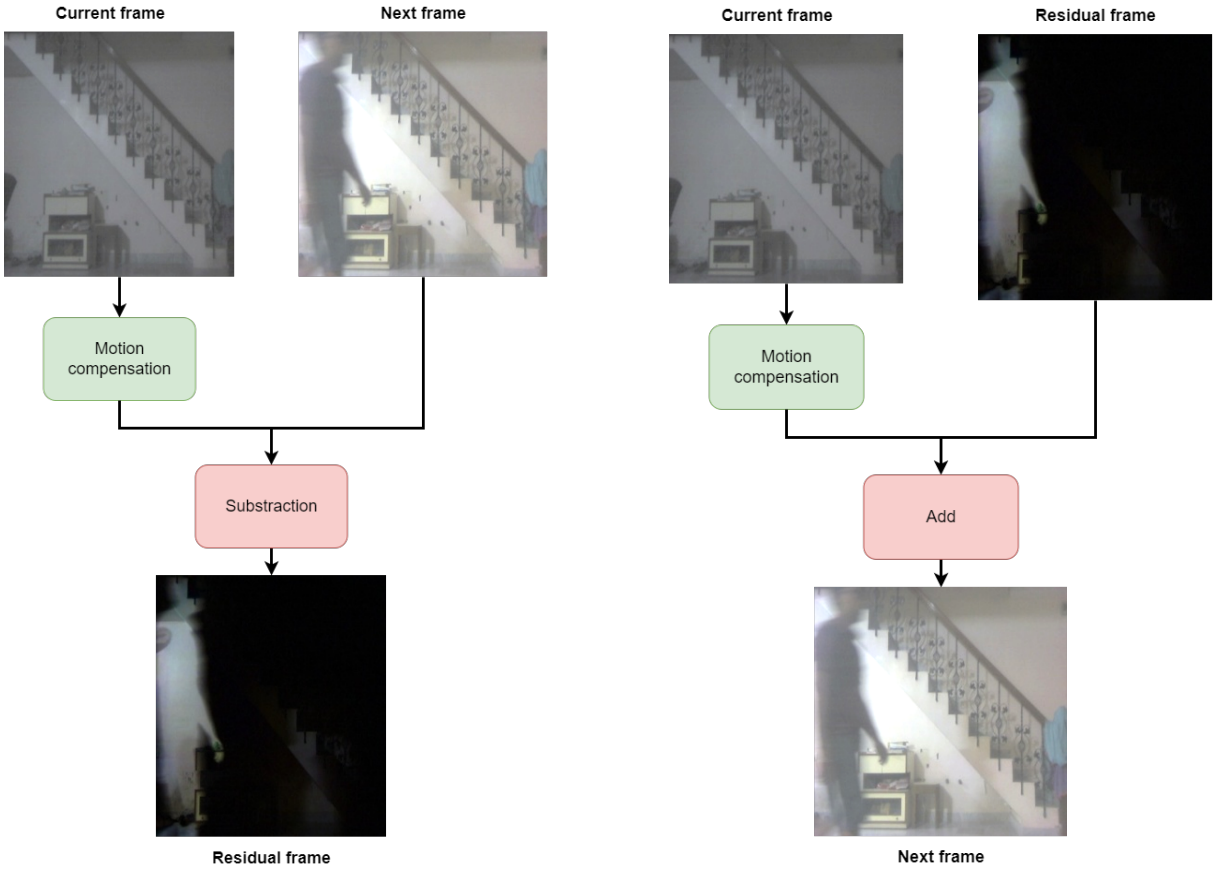


Figure 2.23: Motion compensation [30]

2.3.2.2 Frame differencing

Although motion compensation can greatly reduce the difference between two consecutive frames, it is usually not enough by itself to fully recreate the next frame. Hence, besides motion vectors, one should also save the frame differences between the actual and motion compensated frames. These differences are known as residual frames [20]. Therefore, the decoder predicts each frame by first taking the previous reference frame, then compensating for the motion using motion vectors and finally adding the residual frame (see Figure 2.24).



(a) Encoder part (b) Decoder part

Figure 2.24: Frame differencing [30, 20]

The choice to save residual frames rather than the original frames is far from innocent. Indeed, residual frames have much less information than the full reference frames. Hence, they are highly compressible.

The entire video compression process goes as follows. Traditional video compression algorithms represent a video as a sequence of reference frames followed by residual ones. Thus, there are two types of compression here: intra-frame coding (subsection 2.3.1) and inter-frame coding (subsection 2.3.2). On the one hand, intra-frame coding compresses a frame by throwing out visually redundant information within the frame and storing the rest more efficiently. On the other hand, inter-frame coding achieves a high compression efficiency by exploiting the similarities between consecutive frames.

2.3.2.3 I, P, and B frames and GOP

A final key feature to address in temporal coding is frame type, namely **I frame** (or **I**ntra-coded frame), **P frame** (or **P**rediction frame), and **B frame** (or **B**idirectionally predicted frame). *This subsection is mainly inspired by Digitally Compressed Television [2].*

When an encoder is turned on, all picture memories are empty. The first frame is sampled, and the DCT processor creates the frequency coefficient matrices for the luminance and color blocks of the picture and sends them to the IDCT decoder at the receiver side. The local decoder produces the same image that will exist at the receiver. This image is then compared to the contents of the previous picture memory. Since there was no previous picture, the previous picture memory is empty, and there is nothing to add or subtract. The entire decoded picture is loaded into the picture memory. When the second frame is sampled, the previous picture is subtracted from it. Only macroblocks of the moving parts in the second frame have changed. These macroblocks are encoded as differences from the previous picture, and a new coefficient matrix is transmitted to the receiver side. At the decoder, the coefficient matrix of the moving parts is received and added to the previous image stored in the previous picture memory. In this example, two kinds of coefficient matrices have been transmitted. The first coefficient matrix was a complete picture, called an intra-frame or I frame. The next frame contained information used to predict the contents of the picture memory for display. This frame is called a P frame.

2.3.2.3.1 Multiple I frame requirements One can imagine that only one I frame suffices to decode a stream, with all the rest being represented thanks to P frames. However, there exist a couple of reasons why this is insufficient.

- Firstly, if the viewer misses the I frame while flicking through TV channels, the latter would not get the correct image. In the example described above, the viewer would only get the moving macroblocks and not the rest of the picture.
- Secondly, as previously mentioned, motion compensation attempts to find where each macroblock has "moved" between frames. However, it is unlikely that an exact match will be found, due to, for example, distortion caused by perspective and lighting changes. Therefore, when the best match is found, picture differencing is used to send the changes in the macroblock along with its new position. It is important to recall that the changes are also compressed and thus rounded to reduce the data size. Unfortunately, as the P frames are transferred, imperfections and distortions accumulate. There comes a time when one must reset the process every so often to limit the propagation of errors.

For those reasons, I frames are sent several times a second to allow channel surfers to have a starting point and limit the propagation of errors.

2.3.2.3.2 Bidirectionally predicted frame (B frame) To understand the concept of B frame, let us consider the example in Figure 2.25. In this example, we have five frames of the same landscape but with different positions for the airplane.

When the airplane moves from one position to another, it uncovers a piece of sky, which is present in subsequent images. If, for example, Figure 2.25c was already in memory, one could take advantage of the piece of sky uncovered from future frames but hidden by the plane in the current frame. In other words, one would have the ability to predict backward in time. The ability to predict both forward and backward in time is possible if we introduce a slight delay in the transmission and add memory to the receiver for more than one video frame. The frames that allow bidirectional prediction is called B frames. These frames increase transmission efficiency at the expense of increasing complexity and memory usage on the decoder side. The trade-off in cost and complexity is governed by the memory cost. However, with the cost of memory becoming less and less critical nowadays, B frames are a worthwhile investment since bandwidth is the ultimate scarce resource.

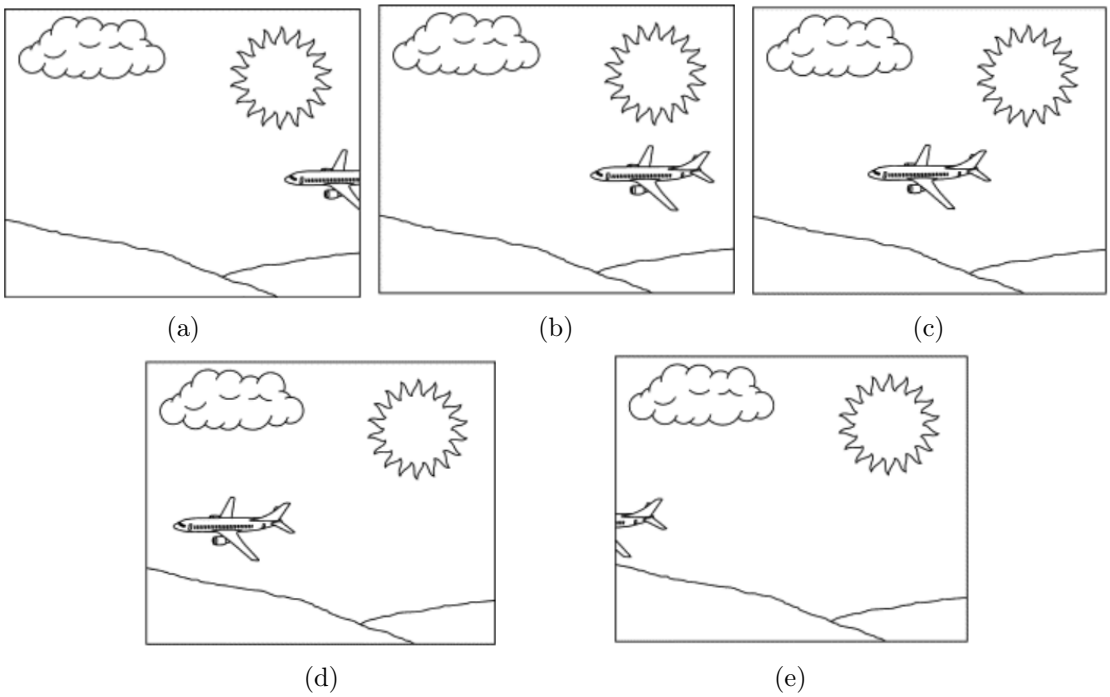
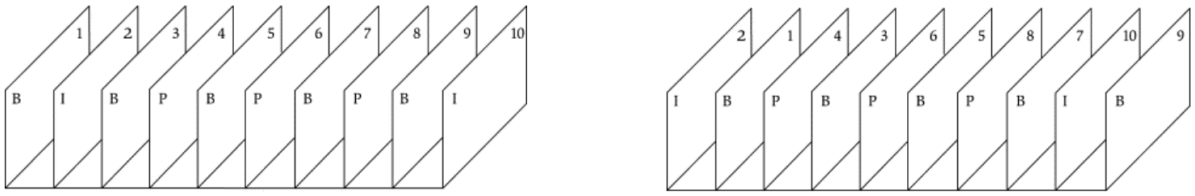


Figure 2.25: B frame [2]

2.3.2.3.3 Group Of Pictures In this subsection, three kinds of frames were defined: I, P, and B frames. The **G**roup **O**f **P**ictures or **GOP** structure specifies the order in which these frames are arranged (see Figure 2.26). Figure 2.26a shows the order in which the frames are displayed while Figure 2.26b depicts the order in which the frames are transmitted. The latter order is different from the display order to minimize receiver complexity.



(a) Display order (b) Transmission order

Figure 2.26: GOP structure [2]

In some sophisticated encoders, the GOP flow is interrupted at a scene change and an I frame is transmitted. As a reminder, an I frame contains all of the information needed to decode the frame itself and thus is not dependent on adjacent frames. However, I frames require much more data than other frames, typically a ratio of 5:3:1 for I:P:B frame size exists. Therefore, the data flow is uneven, and the decoder must have a **buffer**[9] of sufficient size to even out the flow.

2.3.3 Numerical example

To conclude this section, it is interesting to illustrate the discussion with a realistic numerical example of video compression based on both spatial and temporal redundancy [41]. Let us consider a 512×512[10] color video with a byte sample for each color channel at a frame rate of

[9] In computer science, a buffer is a data area shared by hardware devices or program processes. The buffer allows each device or process to operate without being held up by the other.

[10] As a reminder, the first number defines the number of pixels in width while the second number defines the number of pixels in height.

30 **fps**[11]. On the one hand, let us assume that intra-frame coding implies 0.2 **bpp**[12] on average for each color channel. On the other hand, let us assume that inter-frame coding implies 0.02 bpp including overheads for motion vectors for a GOP length of 100 pictures.
Then,

- The uncompressed bit rate is given by

$$512 \times 512 \times 30 \times 24 \simeq 189\,\text{MB/s}$$

- In intra-mode, as all frames are compressed independently, this gives an intra-mode compressed bit rate of

$$512 \times 512 \times 30 \times 0.2 \times 3 \simeq 4.7\,\text{MB/s}$$

- In inter-mode, as 1 in every 100 frames are intraframes and the other 99 are interframes. Hence the inter-mode compressed bit rate is

$$512 \times 512 \times 30 \times \left(\frac{0.6 + 99 \times 0.06}{100}\right) \simeq 514\,\text{KB/s}$$

This clearly certifies that one can achieve significant savings through the use of inter-frame coding to exploit temporal, as well as spatial, correlations. Indeed, the bandwidth is reduced by slightly more than 375 times compared to the bandwidth required for raw data transmission.

2.4 Video coding format

A video coding format is a content representation format for the storage or transmission of digital video content. Therefore, it uses standardized video compression algorithms, as seen in section 2.3. Similar to video formats, there is a multitude of video coding formats.

The notion of video coding formats should not be confused with the notion of a video codec. This confusion is amplified by the term abuse of *codecs*, which is commonly used to refer to video coding formats such as H.264. While, as explained earlier, video coding formats refer to the compressed data, namely the specification, video codecs refer to a specific software or hardware implementation capable of compression or decompression to/from a specific video coding format. `Xvid` is an example of a video codec implementing encoding and decoding videos of the MPEG-4 Part 2 video coding format [70].

Another concept not to be confused with video coding format is the concept of container formats such as MP4 or AVI. The encoded video content is normally accompanied by an audio stream, which is also encoded as ACC-encoded audio (an audio coding format). The two coding formats are then combined in a single format called a container format. Therefore, a container format, also called a wrapper or metafile, is a file format that allows multiple data streams to be embedded into a single file, usually along with metadata for identifying and further detailing those streams. However, the metadata available in the container format is not intended to give instructions for decoding the stream (the role of the video codec) but rather to identify according to which format the data is encoded [70].

[11] fps = frame per second.
[12] bpp = bit per pixel.

The following Figure 2.27 illustrates the history of video compression standards.

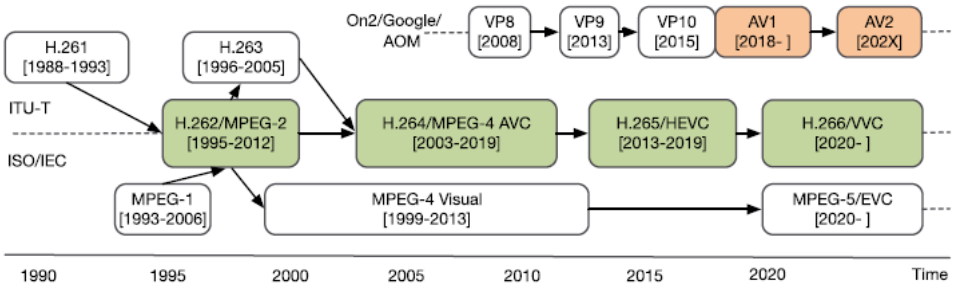


Figure 2.27: Video compression standard history [35]

As explained in *Vcodex: Introduction to Video Coding* [19], one must note that, even though all these standards share the same basic concepts, performance is constantly increasing. Indeed, H.265 is, on average, twice as efficient as H.264, which was already twice as efficient as MPEG-2 on average. The main reason behind this constant evolution in performance is the fundamental trade-off between video quality, computation, and compression rate (see Figure 2.28).

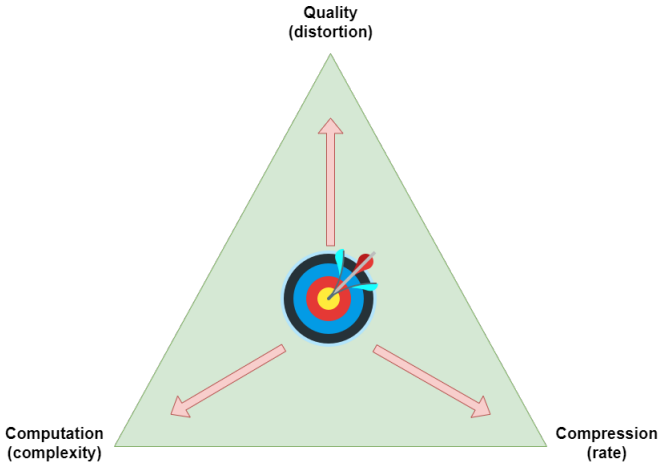


Figure 2.28: Video compression trade-off

In other words, it is possible to achieve better quality, a better compression rate, or even both, if it is possible to increase the computing power. So far, however, computing power is also something that keeps increasing over time. As a result, new compression standards are using increasingly complex algorithms, which significantly improve performance. In the following, we will detail the key features of some of the main formats that are widely used in the broadcasting world based on several scientific books and thesis [13, 3, 36].

2.4.1 MPEG-2

MPEG-2, *a.k.a.* H.262, is a standard for the generic coding of moving pictures and associated audio information created in 1994 [61]. **MPEG** stands for **M**oving **P**icture **E**xperts **G**roup and describes a combination of lossy video compression and lossy audio data compression methods, which permit storage and transmission of movies using currently available storage media and transmission bandwidth. Even if MPEG-2 is not as efficient as newer standards such as H.264/**AVC** and H.265/**HEVC**, it is still widely used since backward compatibility with existing hardware and software means such as over-the-air digital television broadcasting and **DVD**-Video standards are still needed. MPEG-2 implements the traditional floating-point DCT with (8×8) blocks[13]

[13]The DCT blocks are not always the same as macroblocks used in motion compensation.

followed by the quantization and the entropy coding. Besides, MPEG-2 implements motion compensation on (16×16) macroblocks as fully explained in *MPEG-2 video compression* [15].

2.4.2 MPEG-4 Part 2 Visual

MPEG-4 was initially designed to provide video coding solutions at very low bit rates, which resulted in the standard MPEG-4 Part 2 Visual. This standard differs from MPEG-2 and H.264. Therefore, one should note that MPEG-4 alone is ambiguous since MPEG-4 Part 10 refers to H.264. This standard defines the **A**dvanced **S**imple **P**rofile (**ASP**). Unfortunately, MPEG-4 Part 2 Visual was quickly outperformed even in fields it initially targeted by formats such as H.264.

2.4.3 H.264

H.264 or MPEG-4 Part 10 is a video compression standard based on block-oriented coding and motion compensation (see section 2.3 for more details), also known as **A**dvanced **V**ideo **C**oding (**AVC**) [47]. The first approved version was released in May 2003. This format was standardized by the ITU-T **V**ideo **C**oding **E**xperts **G**roup (**VCEG**) of Study Group 16 together with the ISO/IEC JTC1 **M**oving **P**icture **E**xperts **G**roup (**MPEG**). Still to this day, it is by far the most commonly used format for the recording, compression, and distribution of video content. It supports resolutions up to and including 8K **U**ltra **H**igh **D**efinition (**UHD**). This standard was developed to provide good video quality at substantially lower bit rates than previous standards, *i.e.*, half or less the bit rate of H.263 (a poorly known and used predecessor to H.264), without significantly increasing the complexity.

2.4.3.1 Main features

As described in Morgan Kaufmann's book [16], the improvement over other formats of the time is based on features such as:

- A reduced-complexity integer discrete cosine transform, which is a variant of the one developed in subsection 2.3.1.2. Indeed, such a DCT can overcome numerical mismatches between the encoder and decoder, leading to drift.
  Using separability property to obtain a classic 1D 4×4 DCT, one has the following transformation matrix, T:

$$\mathbf{T} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) \\ \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ \sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & \sqrt{\frac{1}{2}}\cos\left(\frac{\pi}{8}\right) & -\sqrt{\frac{1}{2}}\cos\left(\frac{3\pi}{8}\right) \end{bmatrix}$$

  This matrix becomes the following by applying the integer transformation [42]:

$$\mathbf{T}' = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix}$$

- Variable block-size segmentation, as its name suggests, refers to the fact of considering macroblocks of varying sizes such as (4×4), (8×8), or (16×16).
- Multi-picture inter-picture prediction which corresponds to temporal coding (see subsection 2.3.2).

2.4.3.1.1 Profile Another feature of the H.264 format is the use of numerous profiles. An H.264 profile defines what the encoder can use when compressing a video. Since there are lots of H.264 features that the encoder can enable, profiles can be helpful. As a matter of fact, profiles ensure compatibility between devices that have different decoding capabilities. With profiles, the encoder and decoder agree on a feature set that they can both handle. In short, profiles represent the trade-off between compression performance and computational complexity.

As said earlier, there is a multitude of profiles whose complete details can be found on Wikipedia [47]. However, they can be grouped into three increasing categories in order of encoding efficiency: **Baseline**, **Main**, and **High** profiles. Nowadays, high profile is the most used profile for efficiency reasons. Nevertheless, the different profiles each define their own feature set:

- Generally, the **Baseline** profile restricts the encoder to certain basic features only. It supports I and P frames and is dedicated to real-time applications such as video conferencing, or platforms with low processing power.
- The **Main** profile is a superset of the baseline profile. The main profile uses I, P, and B frames and is mainly used for digital non-HD television broadcasts.
- The **High** profile is a superset of the main profile. The high profile offers a higher compression ratio than the others, at a slightly increased implementation complexity and computation cost. It is mainly used for high-definition applications. For example, it is used to store HD videos on Blu-ray discs or for **HDTV** broadcasts.

The profile only indirectly influences the quality. Some features of higher profiles may enable one to get the same quality with lower file sizes as compared to lower profiles. Thus, if the encoder has a specific bit rate to spend, it will be able to create a better quality video because it achieves much better compression at a higher profile. This also explains why one achieved a smaller file size with a high profile. Indeed, the encoder in the high profile can use more advanced compression techniques to create a video file that has the same quality as the baseline profile, but with a smaller size.

2.4.4 H.265 (HEVC)

High-**E**fficiency **V**ideo **C**oding (**HEVC**), also called H.265 or MPEG-H Part 2, is a video compression standard designed as a successor to the widely used Advanced Video Coding. In comparison to AVC, HEVC offers from 25% to 50% better data compression at the same level of video quality or substantially improved video quality at the same bit rate [54]. To achieve such an improvement, HEVC relies on two aspects of compression.

- On one side, in the inter-frame coding, HEVC increases drastically the maximum size of each macroblock. Indeed, the macroblock can range from 16×16 to 64×64 pixels, and larger sizes are more efficient in terms of data compression. These macroblocks have been renamed to coding tree units.
- On the other side, in the intra-frame coding, HEVC increases the number of prediction directions used in intra-prediction. Indeed, while there were only nine prediction modes in H.264, HEVC implements up to 35 prediction modes. The tremendous advantage is that these directions include far greater accuracy when larger block sizes are used, thus generally much higher quality compression.

2.4.5 V8 - V9

VP8 is an open and royalty-free video coding format created by On2 Technologies and owned by Google since 2010. Google's motivation was to avoid paying royalties, and conversely to pass

on these royalties to its customers, on MPEG-2 or H.264 intellectual property. This standard is thus broadly based upon the principles of MPEG-2 and H.264 [17].

In the same vein, VP9 is the successor of the VP8 format and completes the encoding performances by essentially basing itself on HEVC techniques while avoiding paying royalties [72].

2.5 Video over IP

IP video refers to a video distributed on an IP network. Hence, **IP** stands for **I**nternet **P**rotocol. Internet Protocol is a set of standards for communicating over computer networks. Unlike **S**erial **D**igital **I**nterface (**SDI**) video transmission, which was the way of transmission used for years, in video over IP, the media inputs are deconstructed into different streams and then sent over an IP network as individual packets. Video transmission through SDI requires setting up an SDI cable linking two discrete points. Once the information arrives at its destination through an IP network, it is reassembled and aligned using a synchronization technology, known as **P**recision **T**ime **P**rotocol (**PTP**). *This section is inspired by Paul Richards' book [45].*

2.5.1 Advantages

IP video is a real breakthrough in the broadcasting world as it offers many advantages over traditional cable connection options, which will be discussed in this subsection.

2.5.1.1 Scalability

Nowadays, the pressure on the audiovisual world and quality it provides keeps increasing, which means that producers are forced to improve the immersion in the scene illustrated, for example, by increasing the number of cameras. However, when using SDI and HDMI cabling, adding additional cameras and other sources can become complicated. Hardware switchers can quickly run out of inputs, and many computers are limited on the number of capture cards or PCIe inputs that can be connected. With IP video, a single Ethernet cable can handle a large number of sources at no additional cost in terms of capture cards and hardware video switchers.

2.5.1.2 Decentralized distribution

Another advantage of IP video over standard video transmission is the possibility of decentralized distribution. Indeed, with a classical hardware switcher, all sources are routed to one location. Whereas, with IP video, sources are available anywhere on the local network and can thus easily be sent to any other location on the network, as illustrated in Figure 2.29. This enlarges the production and distribution possibilities.

Figure 2.29: Decentralized distribution system [45]

2.5.1.3 Distance barrier

IP video breaks the distance barrier of standard cabling options. Although SDI-type cables are the best in terms of transmission length compared to HDMI or **USB** cables (limited length, about 10 meters for USB and 25 meters for HDMI before extenders must be used), in practice, there is still a certain limit based on the capacity and quality of the SDI cable. Moreover, the Ethernet cable used for IP video can be used, at the same time, for video, control capabilities, and powering a device, while an SDI or HDMI cable can only handle audio and video. Even if there exist limitations as to the length of individual Ethernet cables (about 100 meters), these cables only need to be linked to the nearest router or switch and not connected all the way from the source to the switcher.
Besides, thanks to video over IP, it is now possible to imagine a wireless video transmission system, as shown in the following Figure 2.30.

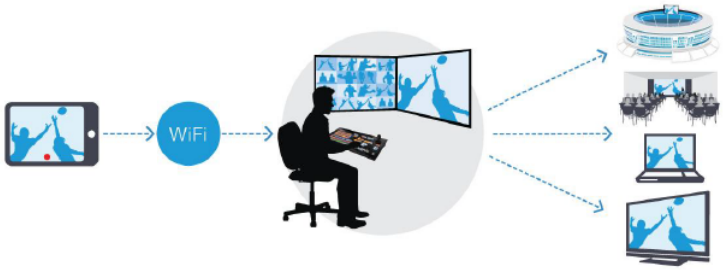


Figure 2.30: Wireless video transmission [45]

2.5.1.4 Affordability

Without a doubt, the most appealing advantage of video over IP for many is affordability. It is well-known that a hardware solution is more expensive than a software solution. Once again, this rule is not missing in this case. Indeed, the prime example is high-quality capture card hardware (a few hundred euros each). Before IP video, all video sources had to be run directly from the source to a capture device in order to be used with a video production solution. Thanks to video over IP, there is no need for expensive capture cards anymore and it is then possible to capture video directly through a computer's network interface. This has made it possible to democratize video production.

2.5.2 IP video considerations

While all these benefits are undeniable and confirm the fact that IP video is a huge step forward in streaming production, there are a few considerations that need to be taken into account before deploying such a mechanism.
Probably the biggest challenge is acquiring a strong knowledge of networking. A simple setup may be manageable for anyone with no networking experience. However, more complex configurations may require a higher level of understanding when it comes to networking.
On the other hand, video over IP requires some minimum requirements for computers (8GB system memory, gigabit Ethernet connection, *etc.*) and for networking equipment (gigabit Ethernet, full throughout switch backplane, *etc.*).

2.5.3 NDI

While video over IP is a general transmission technique, NDI is, among others, the most famous IP video production protocol, besides to being royalty-free. **NDI**, which stands for **N**etwork **D**evice **I**nterface, is a standard enabling compatible products to share video, audio, and data across a **L**ocal **A**rea **N**etwork (**LAN**) at a high quality and low latency. Using refined encoding and communication, NDI permits systems, devices, and applications to identify and communicate bi-directionally with one another over IP and to capture, transmit, and receive multiple streams

of high quality, low latency, frame-accurate video, and audio in real-time.
One should note that when one talks about the NDI stream, it refers to a stream following the NDI protocol. As we saw in this section, this is a method of transmitting video. It, therefore, has nothing to do with the video coding format.

2.5.3.1 NDI, NDIHX, and NDIHX2

As well explained in AVONIC's article [73], NDI, for short, refers to as Full Bandwidth, which is the first version released in 2015. Moving from a transmission system based on SDI cables, which required near-to-zero latency, to the NDI protocol raised questions at the time about the latency this solution would bring. The solution to reducing at maximum latency provided by NDI Full Bandwith was by using **S**peed**HQ** (**SHQ**) encoding, which has very low latency while utilizing higher bandwidths to get its signal across. As explained earlier, another concern when talking about video over IP is the available bandwidth on existing infrastructure. With NDI Full Bandwidth, the network restrictions were rather strong. Indeed, although it was possible to have a solution for a 1 GB Ethernet network, such a bandwidth greatly limited the number of sources connected to a single endpoint.

To correct this problem, Newtek developed NDIHX, released in 2017. This version uses the common compression method H.264 drastically lowering the bandwidth required at the inevitable cost of a slight increase in latency. As a matter of fact, there always exists a trade-off between latency and bandwidth: high bandwidth provides low latency, and low bandwidth will always introduce latency since one needs time to compress and decompress the video.
To get an idea, on the one hand, HDMI 1.4, which, by definition, transmits raw data, utilizes 10.2Gbps[14] for 1080p60[15] video without latency. On the other hand, for 1080p60 streams, NDI Full Bandwidth utilizes about 150Mbps[16] with latencies of approximately 16ms whereas NDIHX can use anything between 1 and 50Mbps with latencies ranging between 80-200ms depending on the equipment used.

To further improve the bandwidth requirements Newtek updated NDI|HX in 2019 with the introduction of H.265 encoding capabilities. This update was referred to as NDIHX2.

To summarize, NDI Full Bandwith and NDI|HX are two complementary methods to transport video across a network. Depending on the infrastructure and demands with regards to latency, one can easily choose between the two.

[14] 1Gbps = 1 GB per second.
[15] 1080p60 corresponds to a 1080p resolution and a 60 fps.
[16] 1Mbps = 1 MB per second.

Chapter 3

Practical part

After having broadly outlined the state-of-the-art related to this thesis, in this chapter, we will study in-depth the different practical points of this work, *i.e.*, the implementation that was the subject of my research. To this extent, we will first focus on the configuration of the tools used during this thesis before tackling the main task: video decoding using the NVIDIA Jetson Xavier AGX platform. This task is divided into several sections including hardware acceleration decoding, visualization of decoded data, and parallel programming decoding. All this analysis is, of course, coupled with a performance analysis of the different decoding solutions implemented.

Besides, one should note that the project implementation can be retrieved via the following link:

`https://mseduculiegebe-my.sharepoint.com/:f:/g/personal/jean-lorys_ossohou_student_uliege_be/EikM8kRaUpJIm6tZqG1YgKQBt8WYJutBz9kbGNn2YGktPw?e=afHpRu`

Moreover, there is also a description of how the application is structured and how to use it through the `readme.md` file.

3.1 Development environment configuration

As mentioned earlier, the embedded world was new to us. Moreover, the health context added a certain complexity to the setting up of an operational development environment since it had to be accessible on the Deltatec site as well as from home. For these two reasons, the development environment setup was the first task, and contrary to what it may seem, this task was not so trivial. To deal with it, Deltatec provided me with everything I needed. Therefore, I had at my disposal a Windows desktop, an **Ubuntu** 18.04 desktop, a Deltatec PCIe card[1] as well as the key piece of this project, the NVIDIA Jetson Xavier AGX platform. Figure 3.1 below shows the development environment on the Deltatec site.

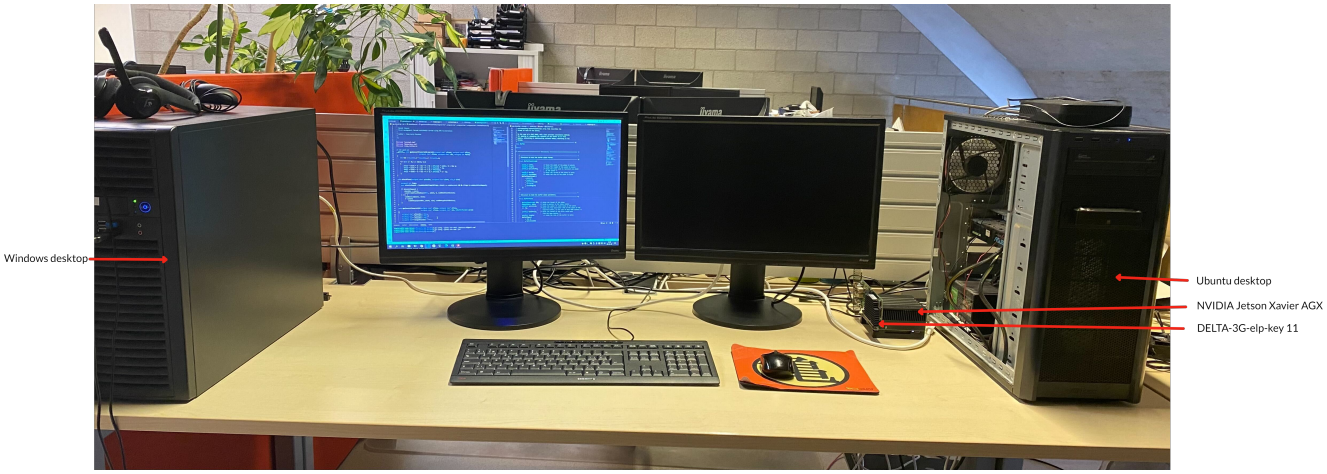


Figure 3.1: Development environment

3.1.1 Windows desktop

The Windows computer was the main machine from which we developed. Therefore, the programs we usually use to program had to be installed, such as **V**isual **S**tudio **C**ode (**VSC**ode), which offers a complete **IDE**[2].

3.1.2 Ubuntu desktop

Although it may seem superfluous, an additional computer to the Windows one was necessary, at least for two specific tasks in the development environment, namely the flashing of the AGX platform [55] and the profiling of parallel programming solutions using `Nsight Systems`[3]. Besides those two tasks, the Ubuntu desktop was rarely used (management of the libraries installed on the platform via NVIDIA SDK Manager).

3.1.3 Flash of the NVIDIA plateform

The initialization of the Jetson Xavier platform was conducted via the Ubuntu 18.01 computer (the version of Ubuntu mattered in the flash process). NVIDIA provides clear guides about the flash. One had to install the NVIDIA Jetpack **SDK** (**S**oftware **D**evelopment **K**it) manager to flash the platform. NVIDIA SDK Manager provides an end-to-end development environment setup solution for NVIDIA's platforms. When it comes to NVIDIA JetPack SDK, it is nothing else than a solution for building end-to-end accelerated applications. JetPack SDK includes Jetson Linux Driver Package with **bootloader**, Linux **kernel**, Ubuntu desktop environment,

[1] See section 3.4 for more details about DELTA-3G-elp-key 11.

[2] **IDE**, which stands for **I**ntegrated **D**evelopment **E**nvironment, is a software application that provides comprehensive facilities (source code editor, build automation tools, a debugger, and others) to computer programmers for software development.

[3] See subsection 3.6.4 for more details about `Nsight Systems`.

and a complete set of libraries for GPU acceleration computing, multimedia, *etc.* Moreover, it also includes samples, documentation, and developer tools for both the host computer and developer kit. Then, one had to follow the steps described in *Install Jetson Software with SDK Manager* [55].

3.1.4 (Tele)working setup

Once all these machines were configured, it was necessary to find an efficient way of working.

When working on the Deltatec site, we used Visual Studio Code, a lightweight but immensely powerful IDE, almost exclusively for development. Thanks to its extremely convenient `SSH`[4] extension, part of the Remote Explorer suite, this allowed us to connect remotely to the NVIDIA Jetson Xavier AGX machine from a Visual Studio Code window opened on the Windows computer. In order to take advantage of such a feature, it was necessary to ensure that OpenSSH, the connection tool used by the `SSH` protocol, was installed and active on both the NVIDIA platform and Windows computer.

When working from home, the same procedure was applied, except that we first had to connect our personal computer to the Windows desktop via Windows Remote desktop that the company made available to us. Consequently, our personal computer was connected to the desktop computer, which was connected to the Jetson Xavier platform via a Visual Studio Code session in remote `SSH`.

Furthermore, the GitLab tool was also useful for version control as well as code review, if necessary, by our company promoter, Julien JEMINE.

One should also noted that it was sometimes necessary to transfer files such as videos to the platform. For this purpose, the `scp`[5] command or, thanks to the multiple connections available on the platform, the simple and efficient use of a USB stick were sufficient.

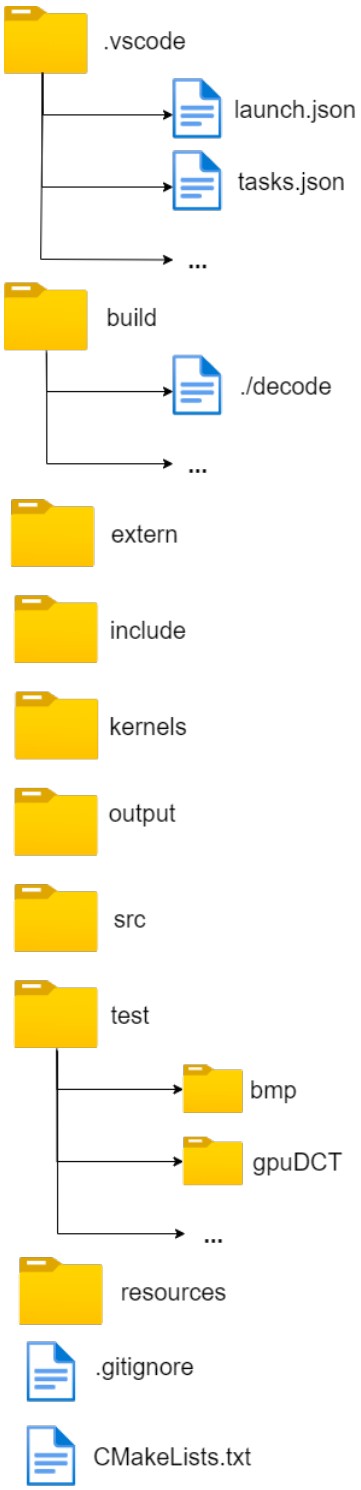
While the NVIDIA team has been careful to provide an incredible development environment with the NVIDIA Jetson Xavier AGX platform, the decoding process is a data-intensive activity. In fact, a one-minute H.264 encoded 1080p video is about 8-9 GB when decoded. However, the actual space available on the platform was only about 10 GB. Therefore, a solution had to be found to store the output somewhere else than on the platform, when the latter was to be tested. For this purpose, we decided to use an external hard disk that we connected to the platform via one of its multiple ports.

[4] `SSH`, which stands for **S**ecure **SH**ell, allows connection to a remote machine (acting as a server) from another machine (acting as a client) via a secure link to transfer files or commands securely.

[5] `scp` is a Linux system command used to copy file(s) between servers in a secure way. The `scp` command, or **s**ecure **c**o**p**y, allows secure transferring of files in between the local host and the remote host or between two remote hosts. It uses the same authentication and security as it is used in the SSH protocol [66].

3.2 General project structure

Before diving into the core of the project implementation, we will first focus on the general structure of the project. The programming language adopted for this project was C++. It is a very complete and flexible language in which we are productive in implementing large-scale projects.



The project file structure illustrated by the figure opposite is characterized by a VSCode solution that includes the different points discussed in the following sections. A VSCode project can be recognized by the `.vscode` folder. This folder is very useful in the development phase since it contains, among other files, the `launch.json` and `tasks.json` files. On the one hand, the `launch.json` file allows configuring VSCode for the debugging of the application. On the other hand, the `tasks.json` file allows configuring tasks in VSCode to run scripts and start processes within VSCode without having to enter a command line or write new code.

Concerning the `build` folder, it is automatically generated by `CMake` when the solution is compiled. This folder contains many folders and files that are automatically generated as well. However, the most important file in this folder is the executable file, `decode`, which has the name of the chosen `CMake` project.

The configuration of a `CMake` project is achieved via the `CMakeLists.txt` file (see `CMake` subsection 3.2.1 for more details). Therefore, this file is a major part of the solution.

The `extern` folder is a folder hosting source code taken as-is from an external source. This folder contains only `CUDA` helpers.

The `include` folder contains all the `hpp` header files of the application. For the sake of clarity and good practice, the header files contain only the documentation and definitions of classes, functions, class variables, enumerations, and others. While the actual implementation of classes, methods, functions, *etc.* is found in the `cpp` files associated with the `hpp` file.

As for the `kernel` folder, it is a folder dedicated solely to parallel programming via `CUDA`. Therefore, it contains the `cuh` header files, which mainly define the specifications of the GPU functions also called kernel[a]. Besides, `cu` files implement all the functions defined in the `cuh` files.

The `output` folder, as its name suggests, hosts all the files generated by the execution of the application.

When it comes to the `src` folder, it contains the actual implementation. In other words, this is where the `cpp` and `cu` files are located.

[a] Kernel in the context of `CUDA` should be confused with an OS kernel.

The `test` folder is the folder related to all the tests carried out during the development of this project. Indeed, as will be further detailed in section 4.1, testing took up a significant part of this project. As a result, there are many sub-folders, one per feature tested. The structure of these sub-folders varies according to the complexity of the test. Indeed, when it is a unit test,

the sub-folder only contains a `cpp` file for the test implementation, a `CMakeLists.txt` file for the solution compilation, and a `.vscode` folder for the test debugging. On the other hand, when it comes to a larger test called an integration test, then the structure is more complete and similar to the one of the general application, although it varies according to the needs. The typical integration test structure is shown in Figure 3.2.
The `resources` folder, as its name suggests, hosts all the files used by the application. In this case, in the context of this application, it is the various videos to be decoded.
Furthermore, as mentioned before, we used version control through Gitlab. Since all the files contained in this project were not source code, it was necessary to exclude files that were not relevant in version control, such as the build folder, videos, application outputs, *etc.* It is for this very purpose that the `.gitignore` files exist. By using regular expressions, it is possible to exclude files from version control.

The structure of a file for an integration test is shown in the following Figure.

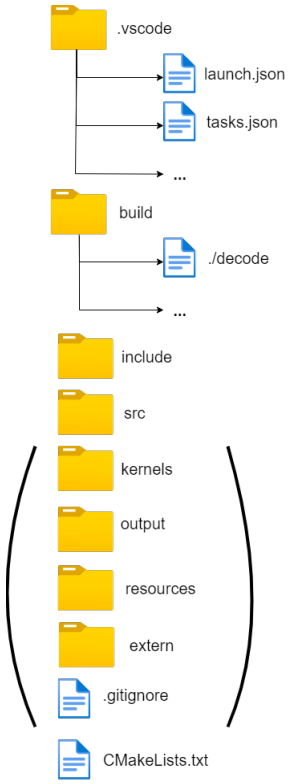


Figure 3.2: Integration test sub-folder structure

It is important to note that the subfolders surrounded by the large brackets are optional. In addition, all the different types of folders have been discussed previously.

3.2.1 CMake

`CMake` is an open-source, cross-platform family of powerful tools designed to build, test, and package software. `CMake` is used to control the software **compilation** process using simple platform and compiler independent configuration files and generate native makefiles and workspaces that can be used in the compiler environment of our choice. `CMake` is configured via a `CMakeLists.txt` file. Unlike traditional `Makefiles`, a `CMakeLists.txt` allows one to compile C/C++ projects in a very compact and easy way. Furthermore, as exploited in this project, it is possible to build several solutions within a single project using `CMake`.
A considerable advantage that `CMake` has over the traditional Makefile system is that VSCode has a very useful extension for `CMake`, which makes building, developing, and debugging the solution much easier.

3.3 NVIDIA Jetson Xavier AGX : Hardware acceleration decoding

The objective of this part of the project was to be able to decompress a video stream coming from a file whose name is given as an argument to the program and which contains frames, coded in different video coding formats (see section 2.4), using the hardware accelerators of the Jetson Xavier AGX module. The application is able to start decoding the file from a certain frame whose index, F, is specified by the user as well as to decode a specific number of frames, N, specified by the user. The program then loads in memory the frames F to $F+N-1$ of the input file, and decompresses the frames in the order R times, with R being another argument of the program, to an output memory area. If a preview is requested via another argument (and only in this case), the application outputs a YUV file containing the decompressed frames. Finally, the program displays the average decoding time for the R decoding rounds.

The interest in the R factor was twofold. On the one hand, it makes it possible to establish a performance average. Indeed, from one decoding to another, the **O**perating **S**ystem (**OS**) can act differently from the program. Therefore, a difference of a millisecond is highly likely. However, this represents a lot in terms of the execution time of the decoding. On the other hand, the fact that the averaging is done internally makes it possible to avoid both the internal overhead of the program, such as initialization for example, and the even greater external overhead produced by the successive execution of the same program. Consequently, the performance estimate obtained internally is more reliable.

3.3.1 Hardware acceleration

Hardware acceleration refers to the process by which an application will off-load certain computing tasks onto specialized hardware components within the system, enabling greater efficiency than is possible in software running on a general-purpose **CPU** alone.

Hardware acceleration combines the flexibility of general-purpose processors, such as CPUs, with the efficiency of fully customized hardware, such as GPUs and **ASICs**[6], increasing efficiency by orders of magnitude when any application is implemented higher up the hierarchy of digital computing systems. A typical example of hardware acceleration is the use of a graphics card for the digital display. Indeed, using a card specially designed for this task allows, first of all, to improve, for instance, the video quality while being more efficient. But it also allows the CPU to be freed up to perform other tasks in parallel. There exist other examples of hardware acceleration, such as AI hardware accelerators (for machine learning, neural networks, and others) and tethering hardware accelerators (for increasing transfer efficiency) [53].

3.3.2 Jetson Linux multimedia APIs

To carry out the practical part of this Master thesis, the understanding and mastery of the Jetson Linux multimedia **API**s[7] key elements were required. Multimedia APIs are a collection of lower-level APIs that support the development of flexible applications by offering more control over the underlying hardware blocks. The Multimedia APIs provide libraries, header files, API documentation, and sample source code for developing embedded applications for the Jetson platforms.

In addition to mastering these APIs, it was also necessary to understand the Linux video API, especially the **I**nput/**O**utput (**I/O**) operations that Jetson Linux multimedia heavily uses.

[6] **ASIC**, which stands for **A**pplication-**S**pecific **I**ntegrated **C**ircuit, is an integrated circuit chip designed for a specific purpose.

[7] **API** stands for **A**pplication **P**rogramming **I**nterface.

3.3.2.1 Video4Linux (V4L)

`Video4Linux`, *a.k.a.* **V4L**, is a collection of device drivers and an API for supporting real-time video capture integrated into the Linux kernel. `Video4Linux` is thus a kind of abstract layer lying between video software and video devices.
This subsection is inspired from Linux documentation [22].

In practice, there exist several kinds of managing I/O operations with Linux devices. In this subsection, we will present the main approach used in the practical part of this thesis, which is based on the second version of the Linux device drivers collection, namely `Video4Linux2` (**V4L2**).

While the read and write operations are the most classical method in I/O operations, as this method is automatically selected to communicate with a device after opening it in V4L2, the I/O operations are mainly based on memory mapping, also called streaming I/O. Indeed, drivers may need the CPU to copy the data in read and write operations, except if they support **D**irect **M**emory **A**ccess[8] (**DMA**). Whereas, only pointers to buffers are exchanged between application and driver in streaming I/O operations, the data itself is not copied. Memory mapping is primarily intended to map buffers in device memory into the application's address space. The device memory can be, for example, the video memory on a graphics card with a video capture add-on.

Conceptually streaming drivers maintain two buffer queues, an incoming and an outgoing queue. They separate the synchronous capture or output operation locked to a video clock from the application, which is subject to random disk or network delays and preemption by other processes, thereby reducing the probability of data loss. The queues are organized as **FIFO**s (**F**irst **I**n **F**irst **O**ut), *i.e.*, buffers will be output in the order enqueued in the incoming FIFO and were captured in the order dequeued from the outgoing FIFO.

Initially, all mapped buffers are in a dequeued state, inaccessible by the driver. For capturing applications, it is customary to first enqueue all mapped buffers, then start capturing and enter the read loop. Here the application waits until a filled buffer can be dequeued, and re-enqueues it when the data is no longer needed. Output applications fill and enqueue buffers. When enough ones are stacked up, the output is started with the `VIDIOC_STREAMON` **I/O control**[9]. In the write loop, when the application runs out of free buffers, it must wait until an empty buffer can be dequeued and reused.

3.3.2.2 GStreamer

`GStreamer` is a pipeline-based multimedia framework that links together a wide variety of media processing systems to complete complex workflows. Therefore, as opposed to Jetson Linux multimedia APIs, this framework provides high-level APIs.
Once this API is installed and using the `gst-omx` plugin on `GStreamer`, the following command allows to decode an H.264 media stream using NVIDIA hardware acceleration.

```
$ gst-launch-1.0 filesrc location=<filename.mp4> ! \
  qtdemux name=demux demux.video_0 ! queue ! H.264parse ! \
  omxH.264dec ! nveglglessink -e
```

However, the objective of this work being to have fine control of the data and the decoding process, `GStreamer` offered us too little flexibility to rely on this framework.

[8] **DMA** is a feature of computer systems that allows certain hardware subsystems to access main system memory independently of the CPU.

[9] An I/O control is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls.

3.3.2.3 CUDA Video Codec SDK

`CUDA` Video Codec SDK offers a set of APIs, including high-performance tools, documentation, and samples for hardware-accelerated video encoding and decoding on Windows and Linux. This solution thus exploits the performance of NVIDIA GPUs on operating systems like Linux or Windows. Consequently, for embedded systems developers, this SDK is not compatible with NVIDIA modules. Moreover, this solution is only related to GPU acceleration, whereas this task aimed to exploit the specific decoding hardware of Jetson Xavier AGX.

3.3.3 Implementation

The decoding application flow follows the following diagram:

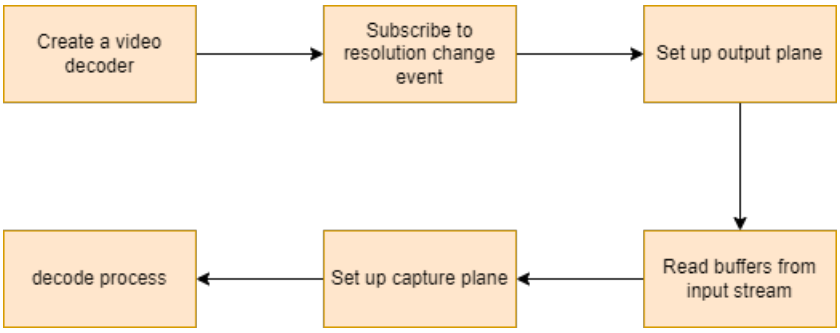


Figure 3.3: Application decoding flow

In the rest of this section, one will review the different important steps of this flow inspired by the NVIDIA decoding samples [44].

3.3.3.1 Decoder creation

This first step consists in creating the `Video4Linux` device allowing it to decode a multimedia stream. The interaction between the Linux decoder and the program is maintained by using the `Video4Linux2` I/O controls and the **file descriptor**[10], given when creating the decoder. Therefore, the file descriptor is of paramount importance in the decoding application and is widely used by the program.

3.3.3.1.1 Thread Nowadays, CPUs are known as multi-core. In other words, a modern computer has one computing unit composed of several smaller ones. This allows a computer to take advantage of parallelism, also called multi-threading. The multi-threading mechanism refers to the ability of a computer to run multiple processes, called threads, in parallel. This mechanism does not necessarily require a multi-core CPU. Indeed, it can be simulated on a single core by switching very quickly between the threads that the computer is running. However, this last method is less efficient than the method with a multi-core CPU. Therefore, the operating system, which works in close coordination with the hardware, has what are called OS-level threads. These are processes that can be launched by the OS in parallel. When the CPU has several cores, there are potentially several OS-level threads running at the same time.

However, there is a slight difference between an OS-level thread and a thread. On the one hand, the OS-level thread is an abstraction provided by the kernel, and its number is limited. Thus, the OS stores the context related to each of its OS-level threads. On the other side, there is no limit to the number of threads. Thanks to the OS scheduling, some threads, abstracted in OS-level thread, are executed by the hardware, or ready to be executed, while the others are waiting to be scheduled.

[10]The file descriptor also noted FD is a unique identifier for a file in UNIX conventions.

3.3.3.1.2 Decoder mode There are two main types of decoders, which differ by the activity that each one has with the threads.

- **Blocking mode**: Creating such a decoder causes the OS-level thread to be deprogrammed on the CPU while waiting for an event to occur. When a thread is deprogrammed, it does not consume any CPU cycles and allows other threads to progress or put the CPU in a lower power state if no other ones are waiting to execute. But it is important to understand that the OS-level thread responsible for the decoder creation is blocked until is run to completion, *i.e.*, no other task can, in the meantime, be scheduled on this OS-level thread.
- **Non-blocking mode**: The creation of such a decoder does not cause the OS-level thread to be blocked but rather immediately returns an error code with `errno`[11], which takes a specific value. The non-blocking mode frees up the corresponding OS-level thread allowing other tasks to be loaded on this thread.

In the context of this application, the creation of the decoder was done in blocking mode. As a matter of fact, a definite advantage that the blocking mode has over the non-blocking one is its simplicity in the protocol to apply.

On the one hand, on a blocking system, if one increases the number of OS-level threads available while increasing the load, throughput increases. However, at a certain point, throughput starts to degrade and latencies start to suffer. This is because, as the number of OS-level threads in the system increases, the context switch[12] overhead start to dominate and each task has to wait longer to be scheduled. In a blocking system, if all programs do only computations, the optimal number of OS-level threads would be the same as the number of logical cores. However, this optimum does not hold if threads also perform I/O since I/O takes significantly more time compared to CPU operations and in such cases, the CPU will idle while most threads wait for I/O.

On the other hand, non-blocking systems allow to perform I/O operations while keeping a minimum number of OS-level threads, which results in maximizing throughput and latency. In addition, the non-blocking mode allows to efficiently use all the computer components, *i.e.*, keeping the CPU active for a payload while waiting for other operations such as I/O to complete [26].

One must note that from the application performance point of view, as long as the computer does not have many other tasks to manage, thus avoiding a boiling plate, the blocking mode allows better performance because, as soon as an OS thread is programmed to advance the execution of the decoding application, it will only be released when its mission is accomplished. Whereas, in non-blocking mode, it is possible that the OS thread previously freed for another task is occupied for longer than the waiting time of the I/O operation requested by the decoding application that interests us here. Indeed, the operating system has a great interest in parallelizing the tasks as much as possible to maximize the hardware use, at the expense (although a trade-off exists) of the latency for a given task. This is why in the context of this project, where the focus is more on the performance of the decoding application itself, the blocking mode is a relevant choice.

3.3.3.2 Event subscription

The subscription to the multimedia event queue will initiate the program. Indeed, this operation is required to catch whenever a resolution change event is triggered to set the format on the capture plane. Thus, the capture plan is only configured if there is a subscribed event in the event queue which will trigger the start of decoding.

[11] `errno` is a global variable accessible from any C++ application and that contains the code of the last error that was triggered in the Linux kernel.

[12] The context switch is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single OS-level thread, and is an essential feature of a multitasking operating system.

3.3.3.3 Output plane configuration

As contradictory as it may seem, the output plane does not provide the output of the decoder. In fact, the output plane must be defined as the plane that hosts the input data, *i.e.*, that reads and stores the data coming from the input. While the capture plane is the plane that processes the input data. In the case of the decoding process, the capture plane is responsible for transforming the encoded data into the raw data. Therefore, the program output corresponds to the capture planes output, as depicted in the following Figure 3.4.

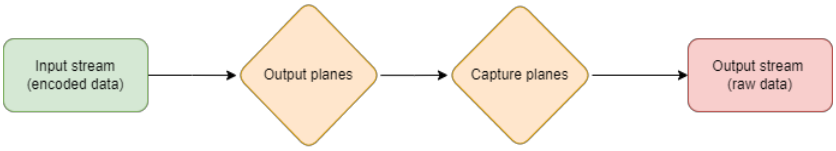


Figure 3.4: Decoding planes

Similarly, in the encoding process, by respecting the definitions given above for the two types of planes, one can see that the output plane handles the input data, *i.e.*, the raw data. While the capture plane transforms the raw data into a video coding format, which is the output of an encoding application, see Figure 3.5.

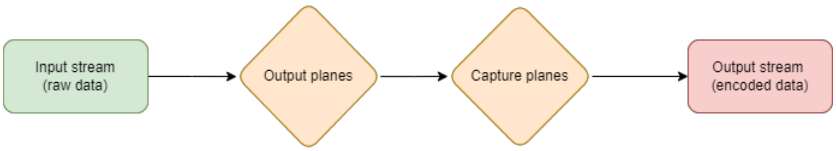


Figure 3.5: Encoding planes

As described above, in the decoding point of view, there are two types of planes: the output plane that handles the reading of encoded data and the capture plane that handles the data decoding. However, the data structure that will hold the data, regardless of type (encoded or decoded), is actually called a buffer. Therefore, the plane can be seen as a shelf that has several glasses (buffer) filled or emptied as the decoding proceeds. There are buffers whose type is dedicated to the output plane and others whose type is dedicated to the capture plane. These buffers are arranged in a queue. Besides, the buffer is composed of a couple of data planes, not to be confused with the capture or output planes. These data planes correspond to the various Y, U, and V planes for the YUV format or the R, G, and B planes for the RGB format. Thus, the number of data planes in a buffer depends on the number of planes in the considered format (generally two or three planes).

After configuring the plane, it should request some buffers of a specific type from the hardware. Once this operation has been completed, it is necessary to map the memory of the buffers' data planes before being usable. Finally, the streaming process must start to enable the hardware and produce a video signal.

The output plane configuration is done by specifying the coding format of the stream (H.264, H.265, or other defined by the V4L2 macros) as well as the size of a chunk. However, depending on the multimedia stream nature, it is necessary to configure the plane input mode: either by considering that the frames are complete (NAL units) or by considering that the frames can be incomplete (read by size of chunks).

NAL refers to **N**etwork **A**bstraction **L**ayer and is a way to clearly delimit each input video frame. There are two ways to pack NAL units: packet-transport system and byte-stream format. In the first system, for example, the **R**eal-time **T**ransport **P**rotocol (**RTP**), the transport system

frames the coded data into different pieces. Hence, it can easily identify the boundaries of NAL units in such a way that extra start code, which is a waste of resources, is not necessary. However, there is no such protocol to separate NAL units in other systems. For example, one wants to store an H.264 file and decode it on another computer. In that case, the decoder has no idea how to search the boundaries of the NAL units. So, a three-byte or four-byte start code, 0x000001 or 0x00000001, is added at the beginning of each NAL unit. They are called byte-stream format. In our case, we are dealing with byte-stream format [10].

It is important to note that the transmission of incomplete frames is only due to transmission errors or file corruption, which can occur for distinct reasons. However, if the decoder allows the reception of incomplete frames, it avoids any interruption in the video decoding when an error occurs. This robustness is, of course, a major advantage as transmission errors are not unusual, especially when dealing with network streams. Therefore, in the context of this task, although this is advanced functionality, it was considered useful to integrate the two modes of operation since the final goal was to decode a stream coming from the network and thus, one with possibly incomplete frames.

3.3.3.4 Data reading

As explained above, it is possible to configure the input mode of the output plane in two diverse ways: complete frames (NAL units) or incomplete frames (chunk). Obviously, depending on the chosen mode, the input stream reading will vary.

First of all, one must note that there are no standards for frame size in a video. Therefore, a video may well contain frames of varying sizes. In addition, in the context of byte-stream format (the format used for this application), there is no information in the metadata that allows a video to be broken down into its frames. The only way to count and process each frame is to detect a common prefix. More precisely, there are two possible prefixes to define the beginning of a frame: 0x00000001 (4 bytes) or 0x000001 (3 bytes).
Let us now look at the two ways of reading the input stream according to the input mode of the output plane: `readNalu` and `readChunk`.

3.3.3.4.1 Read NAL units This type of reading assumes that the frames are complete. First, this operation consists in filling an arbitrarily large buffer to be sure that it contains at least one frame. Once this memory buffer is filled, one of the two universal frame prefixes must be detected. Thus, the data starts to be copied to the target memory buffer (output plane) from the moment the first prefix is detected. The copy operation as well as the execution of the function itself ends when a second universal prefix is detected, as depicted in the following Figure 3.6.

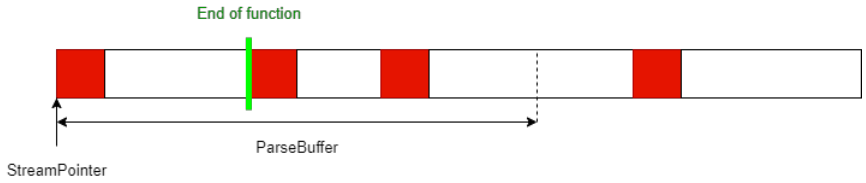


Figure 3.6: ReadNalu

3.3.3.4.2 Read chunks This type of reading assumes that the frames are potentially incomplete. Thus, the function is much simpler since it does not search for any prefix. It reads and fills the output plane with chunks of constant size until the end of the stream (see Figure 3.7). This has the advantage of being faster.

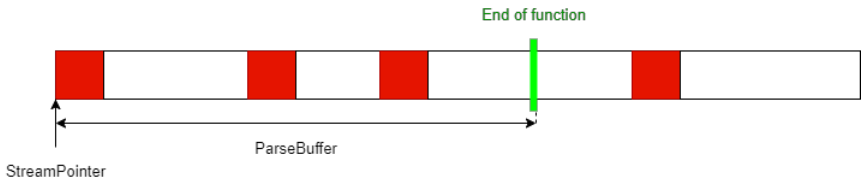


Figure 3.7: ReadChunk

However, as it was necessary to be able to count the decoded frames, this kind of read hardens the frame tracking. Indeed, this function can end right in the middle of a NAL prefix.

3.3.3.5 Capture plane configuration

The capture plane configuration is similar to the one of the output plane. In fact, it involves creating a sort of shelf that contains a set of glasses (buffers) specialized in decoding data that are filled and emptied as the decoding proceeds. However, unlike the output plane, the capture plane uses what is called a destination buffer. Indeed, the encoded data that passes through the capture plane is actually decoded by the hardware. However, the hardware output format does not always correspond with the final format desired by the application and the user (if the interest is to recover the output data). Therefore, the purpose of the destination buffer is to apply the last transformations on the decoded data to respect a certain output format. Besides the format (YUV, RGB, NV), this transformation also considers the resolution or the data layout. It is important to note that these transformations, which involve additional computing time, are only performed if the user wishes to save the decoded data.

3.3.3.6 Decoding process

As mentioned previously, it is the subscription to the event queue that will enable the decoding process to start. The decoding process can be broken down into two main parts: the part related to the output plane and the part related to the capture plane.

3.3.3.6.1 Output plane On the output plane side, it is a matter of reading the encoded data, filling the buffers as it goes along, and inserting the filled ones into the queue so that they can be processed. It should be noted that if there is too much data to be read compared to the number of buffers requested and their capacity, then it is necessary to wait for a signal from the capture plan in order to be able to overwrite the data contained in the buffers already processed. This operation will then be repeated until the end of the input file. Finally, once all the file content has been read, all the allocated buffers are dequeued, which will signal to the capture plane the end of the reading.

3.3.3.6.2 Capture plane On the capture plane side, first of all, it is a matter of queuing all the allocated buffers ready to decode the data. Then, it is necessary to dequeue buffers, full of decoded data, data being in the format specified in the capture plane configuration. Finally, if the user wants to save the program output, it is required to transform this data into the specified format and forward it to the destination buffer.

3.3.3.7 Code structure

In this section, we will introduce the class diagram of the application (see Figure 3.8 below).

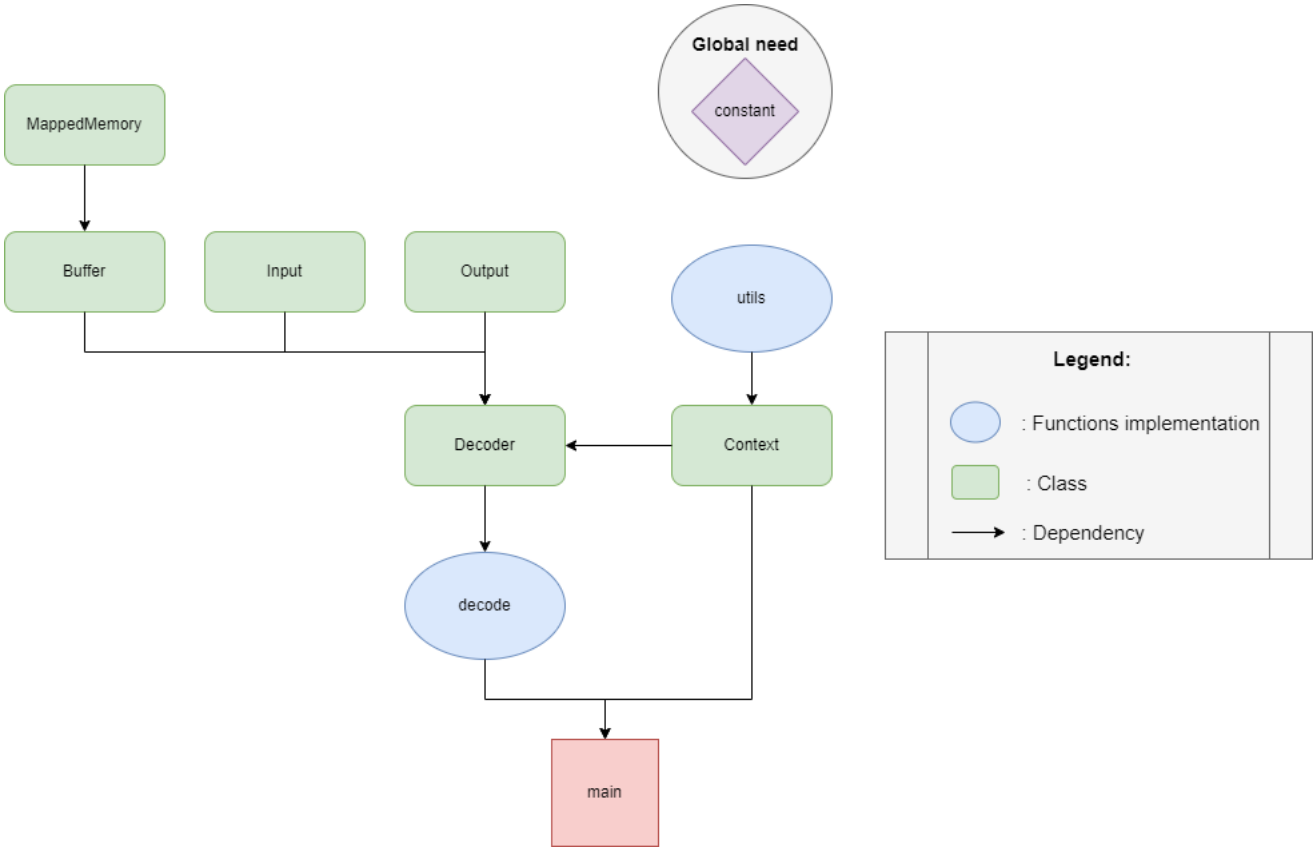


Figure 3.8: Class diagram

Each class/file has a specific task in this application. Let us browse the different elements to better understand how the application works.

- **MappedMemory**: This class handles memory allocation. It enables abstracting the required management when using the `mmap` function from Linux. Therefore, one can allocate some memory and use it instead of using raw pointers and the management it implies (`unmap`)[13].
- **Input**: This class represents the input data source of the program and implements the management related to the input.
- **Output**: This class represents the output data source of the program and implements the management related to the output.
- **Buffer**: This class represents a buffer, as described in subsection 3.3.3.3. It is modeled based on the `Video4Linux` buffer structure.
- **Decoder**: This class represents the decoder. It is, for sure, the most important class of the application responsible, for instance, for the output and capture planes management. It is characterized by its file descriptor heavily used to interact with the Linux kernel.
- **Context**: This class represents the context of the application. It mainly parses the arguments of the application in such a way that the decoder can be rightly set up.
- **utils**: This is a header file that encapsulates all the functions that are not linked to a specific class but are still useful in the context of this project.

[13] See subsection 4.1.2 for more details about memory mapping management in programming.

- **decode**: This is a header file that encapsulates all the functions related to the decoding per se. In short, it manipulates the different methods implemented by the Decoder to handle the decoding of the input stream.
- **main**: This is the entry point of the application.
- **constant**: This is a header file containing all the constants used in this application. The choice of such a file is motivated by the fact that it makes it easy to modify some constants as it can be accessed from a single file.

3.3.4 Results

This work focused primarily on the performance capabilities of the NVIDIA Jetson Xavier AGX module. However, it was also necessary to check that the data was correctly decoded. To do this, a set of small samples of different resolutions and frame rates encoded (in H.264, H.265, and others) and wrapped (in MP4 format) were available. Thus, the developed application was in charge of decoding the different coding format samples whose output was compared with the corresponding container format read. Indeed, MP4 files are easily readable by any multimedia player such as `VLC media player`. It is important to note that the program decodes in YUV format, for which a specific viewer was needed. Some software is available on the internet for free, as is the case for `yuvplayer`.

It is important to note that the execution times reported in the following of this subsection (and more generally in the results of all other types of decoding) are obtained by decoding the streams without saving the decoded data. Indeed, the write operation significantly reduces performance, and its impact depends on the writing medium. To get an idea, for a 1080p video of 598 frames, the average decoding time over ten runs is about 3.12 seconds, while it takes about 7.95 seconds when writing on the internal disk and 22.48 seconds when writing on an external hard disk.

3.3.4.1 Power mode

First, we studied the influence of the different power modes of the NVIDIA platform in the stream decoding. Indeed, as mentioned in section 1.4, the platform has several power modes which give certain additional flexibility in the module's resources allocation. In order to do so, we considered the same stream in the same format (H.264) that we decoded according to the different power modes. The stream is 3840×2160 resolution (4K) with 60 fps in high-profile H.264. One should note that this is a very high-quality standard. The results of this experiment are shown in Table 3.1 below.

Power mode	**MAXN**	**10W**	**15W**	**30W ALL**	**30W 4CORE**	**30W 2CORE**
Time (s)	61.32	63.67	62.7	61.71	61.59	62.3

Table 3.1: Power mode influence

Times reported in the Table here above are averaged over five runs.

A first observation is that for such a quality standard (4K with 60 fps), the decoding time is approximately the same as the video time, even using the specific hardware dedicated to decoding. In other words, it takes one minute of decoding for one minute of video. This does not prevent data visualization in real-time, but this limitation must not be neglected. Indeed, for an optimal user experience in terms of streaming, it would be necessary to foresee a minimum loading time to ensure constant decoding ahead of the visualization.

Besides, one can see that the power mode of the platform logically influences the decoding time. Indeed, the more resources the NVIDIA Jeston Xavier AGX module uses, the faster the

decoding. However, one can notice that the variations are quite slight. Indeed, when the platform is at its maximum capacity, the decoding time is 61.32 seconds, whereas, when the platform uses its resources to the minimum, the decoding time is 63.67 seconds, which represents a small difference of just over 2 seconds.
This can be explained by the fact that the solution depends mostly on the decoding specific hardware, which is quite uncorrelated with the configuration of other components such as the CPU or GPU.
For the following experiments, we used the maximum power mode, *i.e.*, MAXN.

3.3.4.2 Coding format

Then, we studied the decoding times for the different reference codecs. One should understand that a comparison between all these times is not relevant since each video coding format has its own standards. Therefore, the same video encoded in H.264, VP9, or MPEG-2 will not have the same quality depending on the chosen format (see Appendix B). Nevertheless, to obtain the times shown in Table 3.2, we have decoded the same raw data video encoded using `FFmpeg`[14] in different video coding formats. The raw data was stored in a `.yuv` file in 4:2:0 for the chroma subsampling and of 1080p resolution.

Coding format	**H.264**	**H.265**	**MPEG-2**	**MPEG-4**	**VP8**	**VP9**
Time (s)	7.39	5.07	166.48	6.19	6.4	5.71
Profile	High	Main	Main	Simple Profile	/	Profile 0

Table 3.2: Decoding time *w.r.t* coding format

Times reported in the Table here above are averaged over ten runs.

One can notice that for a decent quality (1080p) the decoding is pretty fast. Indeed, decoding takes only about 6 seconds on average[15] for a one-minute video, which is remarkable.

Besides, one can note a significant difference in decoding time for the MPEG-2 format compared to the other ones. This difference highlights a certain limitation in the hardware solution. Indeed, it is much more difficult to design a highly flexible hardware solution allowing to accelerate all the existing video coding formats compared to a software solution. It seems that the NVIDIA team decided to put the decoding performance of the MPEG-2 format aside and concentrate on those of other ones, such as AVC or HEVC. However, this choice is not without meaning since, as a reminder, the MPEG-2 format is an old one that was widely used in the past, notably in CD-ROMs.

3.3.4.3 Resolution

It is also interesting to study the influence of video resolution on decoding times. To do this, we considered the same one-minute video encoded in a single format (main-profile H.264) in several different resolutions with a constant frame rate of 30 fps. The results of this experiment are shown in the following Table 3.3.

Resolution	**1280 × 720**	**854 × 480**	**640 × 360**	**426 × 240**	**256 × 144**
Time (s)	3.94	3.44	2.87	2.81	2.64

Table 3.3: Video resolution influence

Times reported in the Table here above are averaged over ten runs.

[14] See subsection 3.5.1 for more details about this library.
[15] For the more sophisticated codecs excluding the MPEG-2 coding format in the average.

As one might expect, the higher the resolution, the longer the decoding time. However, thanks to the figures in Table 3.3, one can see that going from the lowest resolution (256 x 144 also called **QCIF**) to the highest resolution (1280 x 720 also called 720p) studied here does not even double the decoding time, whereas, in terms of raw data quantity, the highest resolution video (4.63 GB) is almost 25 times larger than the lowest resolution video (189.84 MB). As a result, the decoding time is fortunately not a linear function of the amount of data processed. Once again, one can admire the power of the coding and decoding algorithms studied in large part in the second chapter of this thesis and the parallelism capabilities introduced by the decoding hardware. These drastically limit the computation time.

3.3.5 Conclusion

Hardware acceleration is the fastest conceivable solution via the NVIDIA Jetson Xavier module for the stream decoding process, as evidenced by the above-mentioned performance. However, it is important to note that the solution is very specific to the platform on which the program is running. Therefore, the implemented solution would not give the same performance on other platforms, or even worse, it might not be deployable on other systems.

In order to gain portability, at the cost of a performance discount, we also implemented a more portable solution during this thesis. Firstly, a solution that relies exclusively on CPU programming (see section 3.5), and secondly a solution that relies on GPU programming using a parallel programming platform (see section 3.6).

3.4 NVIDIA Jetson Xavier AGX : real time decoding stream viewing via Deltatec PCIe card

An important part of the project was the visualization of the decoded multimedia stream. Indeed, in the first part of the thesis, the decoding correctness was checked through software. Therefore, it was necessary to wait for the decoding of the file to complete, store it, and then only visualize it.

However, as the objective of this thesis was to decode one or more multimedia streams from the network, this procedure had two main limitations. On the one hand, a stream is potentially infinite, which makes it impossible to wait for the end of decoding before viewing. On the other hand, storage is expensive: little space is available on the NVIDIA Jetson Xavier AGX platform, and performance is slowed down by writing to a file.

Therefore, the main objective of this second task was to solve this problem. To solve it, we had to find a way to visualize the decoded stream in real-time. To do so, we opted for a PCIe card manufactured by Deltatec called DELTA-3G-elp-key 11 and programmed with `VideoMaster API`, which is the API also produced by Deltatec associated with their boards.

The purpose of this board was to transfer substantial amounts of data at high speed in order to visualize them through an HDMI screen. As mentioned earlier in section 1.4, this was possible thanks to the numerous connectors of the NVIDIA platform, including the PCIe connection, which interested us in this case.

3.4.1 DELTA-3G-elp-key 11

The following Figure 3.9 illustrates the PCIe card used during this thesis.

Figure 3.9: DELTA-3G-elp-key-11 Deltatec card [49]

As explained in the Deltatec manual [49], this card belongs to a family of high-speed real-time SDI and **PC** graphics mixing PCIe adapter boards. Each connector is able to manage the transmission or the reception of 3G, HD, or **SD** SDI signals. Connectors can even be configured for transmission or reception. The boards perform bus master DMA transfers, with burst mode capabilities, in order to off-load the CPU and to take optimal advantage of the PCI bus bandwidth. The DELTA-key mainly operates in a genlocked environment and may lock itself onto either a reception channel or on an external analog black burst signal. Besides, one can also operate the DELTA-key in free-run mode, essentially for development purposes. DELTA-3G-elp-key 11 has a monitoring HDMI output which is particularly important in the realization of this task. Each HDMI output can be configured to duplicate either the corresponding SDI reception channel or the corresponding SDI transmission channel. Two video packing schemes are available for HDMI output: YUV 4:2:2 or RGB 4:4:4.

3.4.2 VideoMaster SDK

As previously mentioned, `VideoMaster SDK` is a software development kit allowing communication with Deltatec video cards to design applications such as video servers, signal analyzers, or even video processing. This SDK includes device drivers, an API to control the hardware, documentation, and sample source codes. `VideoMaster API` exposes all the hardware functionalities through series of C functions, structures, and enumerations. Using the appropriate wrapper, it is possible to use the `VideoMaster API` through other languages like C++, C#, and others. The API is uniform over Windows, Mac, and Linux and offers hardware abstraction to handle any Deltatec PC devices. To achieve this, `VideoMaster API` implements five main concepts [49]:

- **Handle**: when working with `VideoMaster`, all logical objects are operated through handles.
- **Board**: this logical entity is used to configure and monitor the underlying hardware at the card level.
- **Stream**: this logical object is used to configure, monitor, and operate data transmission and reception.
- **Property**: all configuration and monitoring parameters are abstracted by the way of readable and writeable properties, attached to given board or stream handles.
- **Slot**: when dealing with data exchange from and to the card, one slot abstracts one temporal unit of content.

3.4.3 Implementation

As a reminder, the project was initially broken down into several parts. However, it was necessary to implement each task in a modular way to reuse what was implemented in the previous tasks for the following ones, until the end of the project. As far as the class diagram is concerned, it is, therefore, normal that it did not intrinsically change, except for the addition of a class, HDMI, proof that the previous tasks were cleverly implemented. It should be noted that the class diagram does not reflect the whole application, since from one task to the next, new class methods appeared. The class diagram for the present task is shown in the following Figure 3.10.

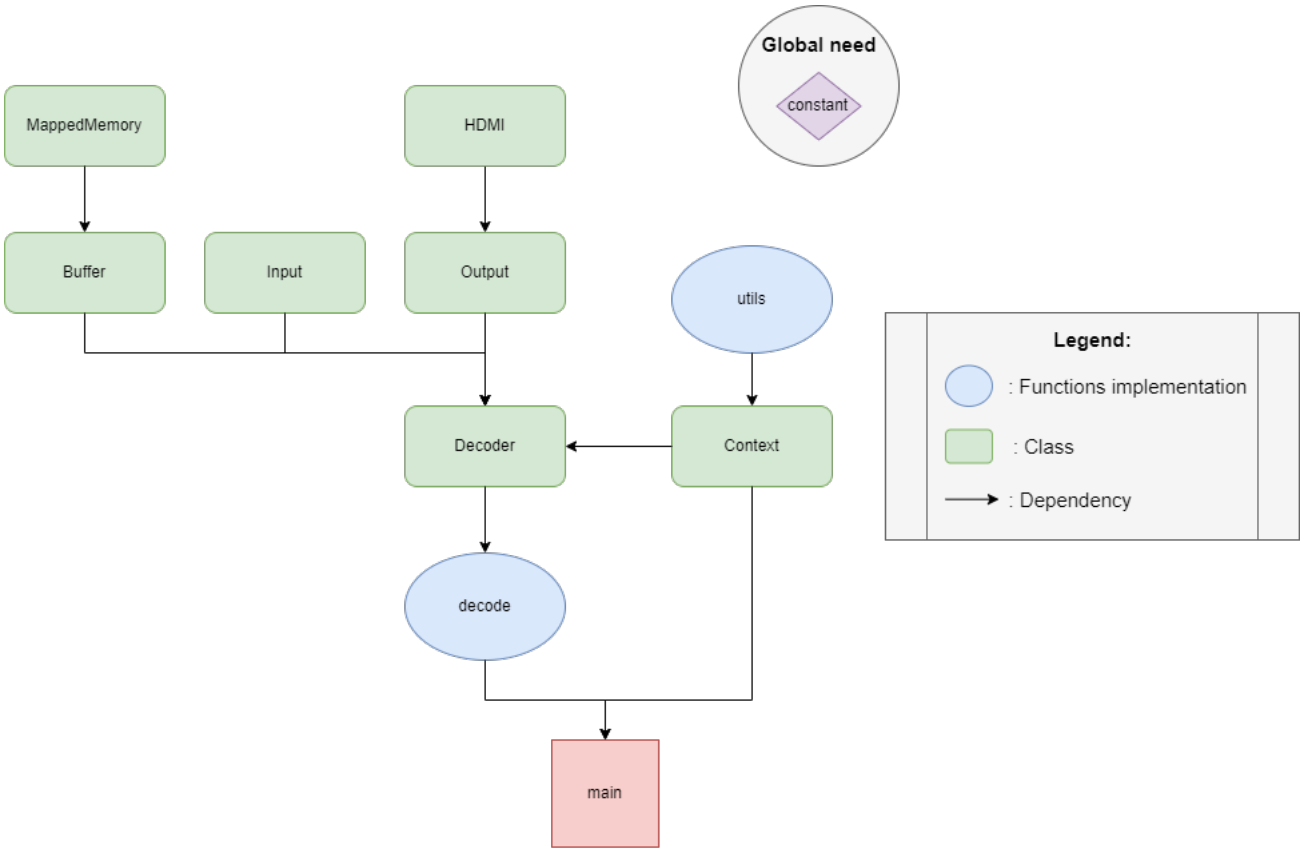


Figure 3.10: Class diagram

The HDMI class is inspired by `VideoMaster` samples [49] and uses the `VideoMaster API` related to Deltatec cards (FPGAs) in order to output the stream via HDMI[16].

In short, when the user specifies the HDMI option, the application opens the board and stream handles and configures them, as well as the HDMI output, to monitor the transmission (TX) stream. Finally, the application launches a transmission thread that handles the data transfer from the decoder to the PCIe card until no frames are left.

3.4.3.1 Data handling from Linux decoder to PCIe card

As explained in section 3.3, one must note that video data is stored in buffer structure as data planes. However, the Deltatec PCIe card used in this project only supports interleaved data of a specific format (YUV 4:2:2 was the chosen format among the ones supported[17]). Therefore, the decoded data retrieved from the NVIDIA module hardware had to be transformed, as depicted in Figure 3.11. A square in the gray 1D table represents the luminance value for the pixel located in row i (specified by the Arabic numeral) and column j (specified by the Roman numeral) of a

[16]For more information about the other classes, refer to section 3.3
[17]See subsection 3.4.1

frame of size 4×4 pixels. Since chroma subsampling is used here, there is a 2:1 ratio of luminance to chrominance values. This is why the U and V arrays are half the size of the Y array.

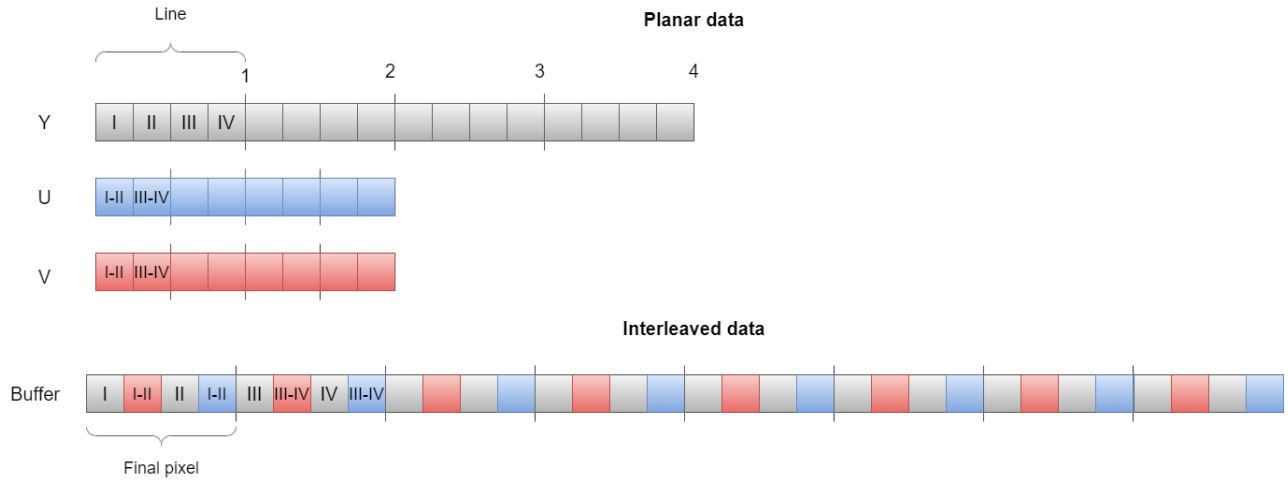


Figure 3.11: Data transformation

It should be noted that even though the stored data represents 2D data, the storage is done in 1D arrays to optimize resources and memory access.
One solution for this data transformation is to rely exclusively on the CPU. Nevertheless, this process is completely parallelizable, and therefore, one might think that it can be programmed on GPU to increase the performance by using parallel programming. However, as will be illustrated in section 3.6, GPU programming does not improve performances due to the overheads introduced by the parallel programming platform. As a result, exclusively using the CPU for this data transformation was the best solution.

In order to better understand how the decoding application works, the following diagram 3.12 summarizes the path followed by the data and the states in which they are located.

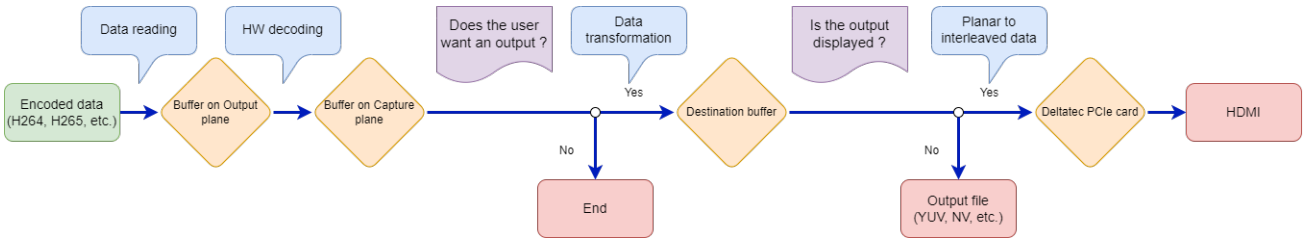


Figure 3.12: Data path and states

3.4.4 Results

In this subsection, we will analyze the time required for data interleaving. Data interleaving is apparently a rather time-consuming process, as it requires going through all the decoded data and processing them one by one to display them. It is therefore interesting to evaluate the workload associated with this processing. In the case of a five-minute video encoded in high-profile H.264 with a frame rate of 25 fps and a resolution of 1080p, the processing to interleave the data takes 5.89 ms per frame on average over three executions. However, one should note that this processing time will vary linearly with the amount of data to be processed. Therefore, for lower resolutions (*e.g.* 720p), the computation time will be lower and vice versa.

Furthermore, we noticed that some packets were dropped in the data transmission from the Jetson Xavier AGX to the HDMI display via the Deltatec PCIe card. We, therefore, decided to conduct a small study concerning the importance of the lost data (see Figure 3.13). To do this,

we decoded and displayed ten times the same one-minute video (1080p at 25 fps) while studying the number of packets lost in the transmission[18].

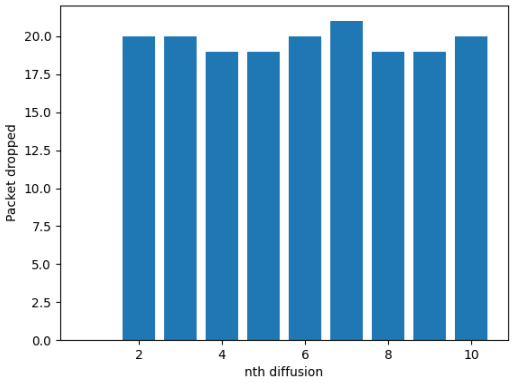


Figure 3.13: Statistics about packet dropped

It turns out that the number of dropped packets is globally constant, whatever the length of the video. According to the tests carried out, 20 packets, on average, are dropped per transmission. In addition, one can also note that a loss never occurs when the video is the first broadcast. This actually means that the loss occurs when the context related to the decoder has to be recreated and, in the meantime, the HDMI output is waiting for frames that do not arrive. Therefore, the PCIe card is forced to drop some slots until the context is ready again. Consequently, this means that this rate, which is already very low in the case of a one-minute video (1%), becomes increasingly low and negligible as the video processed becomes longer. Moreover, discarded packets do not represent actual missed video data but the number of frames that could have been transmitted during the time the decoding context is recreated.

Nevertheless, one solution to this problem would be to reduce the need to create a context for each loop. In other words, only the stream should undergo an operation that is to rewind the video sequence.

3.5 NVIDIA Jetson Xavier AGX : portable solution

In addition to an extremely fast multimedia stream decoding solution like the hardware solution, it was important to produce a solution that was as portable as possible and analyze the performance of this solution when running on the NVIDIA module. To do this, the first alternative to portability was to build a CPU-only implementation. Thus, the solution would be executable on any machine.

However, the streaming world being a very mature and therefore very complex field, the goal was not to reinvent the wheel. Hence, the CPU solution of the decoder uses a reliable and very efficient decoding library called `FFmpeg`.

3.5.1 FFmpeg

`FFmpeg` is an open-source library and, without a doubt, the reference CPU-based multimedia solution used in many applications. This library is able to decode, encode, **transcode**, **mux**, **demux**, stream, filter, and play pretty much anything that humans and machines have created. It supports the most ancient formats up to the cutting edge. In addition, this library is highly portable as it is capable of running on operating systems such as Linux, Mac OS X,

[18]The number of lost packets is an information given by the `VideoMaster API` thanks to a function named `VHD_GetStreamProperty`.

Microsoft Windows, and even on machines such as NVIDIA Jetson Xavier modules. It contains `libavcodec`, `libavutil`, `libavformat`, `libavfilter`, `libavdevice`, `libswresample`, and also `libswscale` which can be used by applications. It is precisely these libraries that we used to implement the solution for this task. On the other hand, `FFmpeg` contains `ffmpeg`, `ffplay`, and `ffprobe` which can be used by end-users for transcoding and playing.

3.5.2 Implementation

The decoding based on the `FFmpeg` library proceeds as follows (Figure 3.14).

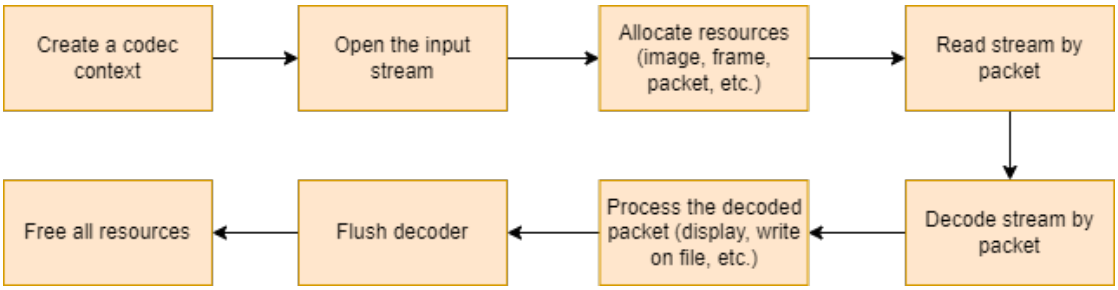


Figure 3.14: FFmpeg decoder workflow

Unsurprisingly, this workflow is similar to the one studied previously (section 3.3).

When it comes to the general structure of the class diagram, it is illustrated in the following Figure 3.15.

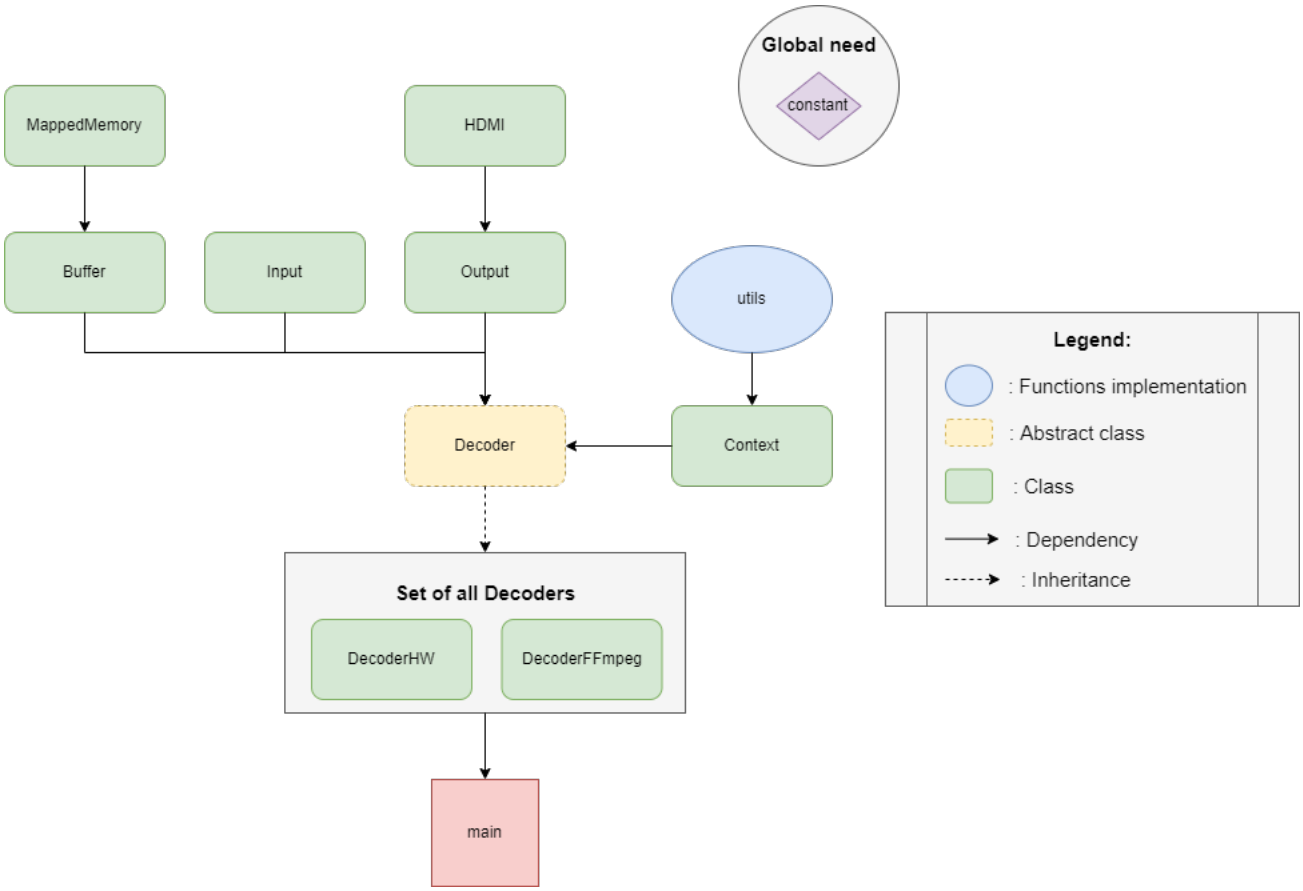


Figure 3.15: Class diagram

As the application hosts several types of decoders, all of which are quite similar in the procedure to be adopted, it was deemed relevant to implement an abstract parent class, `Decoder`,

capturing the essence of a decoder in this project. Thus, any new decoder had to inherit from this class and, consequently, respect a certain guideline in the decoding. Indeed, all decoders had to implement methods such as `init` and `decode` and inherit class variables such as an input (class `Input`), an output (class `Output`) and also the decoding duration. This was to facilitate both the integration of any new decoder and the maintenance of the code.
The `DecoderHW` class implements decoding using the hardware acceleration of the AGX module, while the `DecoderFFmpeg` class implements decoding via CPU, based on the `FFmpeg` library.

One can also notice the deletion of the header file, `decode`, see sections 3.3 and 3.4, whose role was absorbed by the `Decoder` class.

3.5.3 Results

In this subsection, we will repeat exactly the same experiments performed in subsection 3.3.4 using the same video resources. Thus, we will be able to better compare the hardware solution with the CPU solution.
Although this comparison should be taken with a grain of salt, in this subsection, we will also compare the decoding time of the `FFmpeg` solution when it is running on the NVIDIA module on the one hand and a laptop computer[19] on the other hand. This comparison is less relevant since it is a comparison between a computer and the one for an embedded system which is generally less powerful. Indeed, the power supply, the hardware components, and the goals pursued do not correspond. However, it is still interesting to have an order of magnitude to realize how powerful the AGX module is and how it acts as a mini personal computer.

3.5.3.1 Power mode

The results of the experiment related to the power mode influence on the decoding time can be found in the following Table 3.4.

Time (s)	**Power mode**					
	MAXN	**10W**	**15W**	**30W ALL**	**30W 4CORE**	**30W 2CORE**
CPU	126.07	284.54	284.36	280.23	184.48	138.67
HW	61.32	63.67	62.7	61.71	61.59	62.3

Table 3.4: Power mode influence

Times reported in the Table here above are averaged over five runs.

A first interesting result is that whatever the power supply and configuration of the NVIDIA platform, the solution based on the hardware specifically dedicated to the decoding of the module is at least twice as efficient as the CPU-based solution. The hardware-based one is even almost 4.5 times more powerful when the Jetson Xavier AGX is configured at its minimum capacity, *i.e.*, powered with 10 Watts.

On the other hand, as can be seen in Table 3.4 above, decoding times fluctuate greatly depending on the power mode of the NVIDIA platform. Unlike the hardware solution, the CPU one is therefore strongly influenced by the platform configuration. Indeed, there exists a performance ratio of about 2.25 between the most powerful and the least powerful configuration.

Moreover, it is interesting to note that the use of an AGX module was still less efficient than a modern personal computer in terms of decoding time. Indeed, by conducting the same

[19]The laptop computer on which the performance tests were carried out was an HP with an 11^{th} generation Intel i5 processor, 4 cores (8 logical processors) with a computation rate of 2.42 GHz, a RAM of 16 GB, and an Intel(R) Iris(R) Xe Graphics GPU

performance test with the `FFmpeg` library-based solution on an ordinary computer, the decoding time of the 4K video is, on average, 26.21 seconds, which is a bit more than twice faster than the hardware-based solution running on the NVIDIA module with an optimized configuration.

3.5.3.2 Coding format

For the second experiment, Table 3.5 below shows the decoding times for the different reference coding formats.

Time (s)	**Coding format**					
	H.264	**H.265**	**MPEG-2**	**MPEG-4**	**VP8**	**VP9**
CPU on AGX	23.57	77.5	5.15	6.91	9.22	28.54
HW	7.39	5.07	166.48	6.19	6.4	5.71

Table 3.5: Decoding time *w.r.t* coding format

As expected, one can see that the hardware-based solution is always faster than the CPU-based solution except for the decoding of MPEG-2, which is much slower than for the other video coding formats. In addition, one can notice that the performance gain when switching from a CPU solution to a hardware solution is significantly greater when considering the more sophisticated coding formats (H.265, VP9, and H.264). This finding is all the more comforting as these are the most widely used video coding formats today.

3.5.3.3 Resolution

The influence of the resolution on the decoding time is quantified in the following Table 3.6.

Time (s)	**Resolution**				
	1280 × 720	**854 × 480**	**640 × 360**	**426 × 240**	**256 × 144**
CPU on AGX	11.43	5.07	2.78	1.19	0.43
CPU on PC	2.26	1.06	0.57	0.24	0.09
HW	3.94	3.44	2.87	2.81	2.64

Table 3.6: Video resolution influence

It is important to note that the difference between the two solutions in terms of video resolution also depends on the video coding format chosen, as we saw in Table 3.5. However, it was felt relevant to carry out this analysis with the H.264 format as this is the most widely used format in the audiovisual industry.

As before, we can logically observe that the higher the video resolution, the longer the decoding time with both hardware- and CPU-based solutions.
Furthermore, one can see that the hardware solution is better than the CPU solution up to a certain point. Indeed, for very small resolutions, the CPU solution offers better performance. Remember that standard broadcast resolutions are usually at least 480p. In this respect, opting for the hardware solution will always be better in practice.
On the other hand, we can also note that, contrary to the time evolution for the hardware solution, which rather follows a kind of logarithmic function, the evolution of the decoding times for the `FFmpeg` solution is quasi-linear, as shown in the following Figure 3.16.

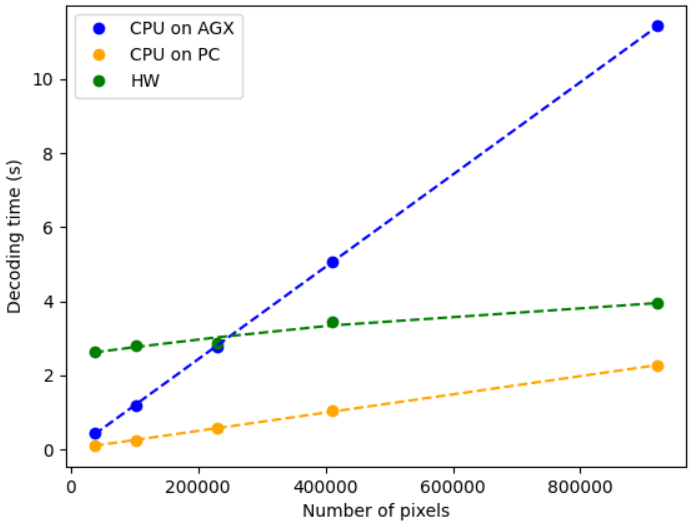


Figure 3.16: Decoding time evolution

A plausible hypothesis to explain this phenomenon would be the existence of an overhead linked to the use of decoding-specific hardware, which would add a constant cost to the execution time while doubling the data to process simply doubles the processing time when it comes to the CPU.

3.5.4 Conclusion

As we expected, the performance of the CPU solution is much lower than the performance of the NVIDIA platform hardware acceleration solution in almost all cases. However, the performance loss is compensated by the gain in the solution portability, which was the goal of this task. It is clear that there exists a trade-off between performance and portability. In the next section of this work, we will study the possibilities to improve the performance of the decoding solution without affecting (too much) its portability.

3.6 NVIDIA Jetson Xavier AGX : parallel decoding

As we saw in the previous section, the portability of the decoding solution has a cost. In this thesis, we studied alternatives to decrease this cost to portability. In other words, the goal of this task was to provide a solution that is both as efficient as possible and as portable as possible.

It is important to note that decoding is a task of a parallelizable nature, which means that it is possible to split this task into smaller ones, each handled by different computing units. This characteristic is far from being intrinsic to any task. Indeed, to make it possible, the problem must be easily split up, and each part must be independent of the others. Fortunately, this characteristic is not binary, and every task has its degree of parallelization. At present, when the growth of the power of a single computing unit tends towards a practical limit, the world of computing takes advantage of parallelism to drastically reduce computing times.

As we saw in subsection 2.3.1, a key step in the decoding process is the discrete cosine transform. As a reminder, this step consists in dividing each frame into macroblocks, each of which can be processed independently of the others. Of course, this characteristic can be exploited by parallelism, which is the objective of this task. Indeed, this task consists in speeding up, thanks to parallel computing, some parts of the decoding in order to improve performances without significantly affecting the portability of the solution. Nevertheless, although the decoding

task lends itself well to parallelism, decoding is not entirely parallelizable. Indeed, macro blocking the data, splitting the stream into NAL packets, and other decoding steps are not easily parallelizable processes, if at all.

Moreover, in this section, we will also discuss the acceleration of the previous task (see section 3.4) via parallel programming. Indeed, as said before, data interleaving is a parallelizable task, and we therefore get acquainted with parallel programming by also implementing GPU acceleration of this process.

3.6.1 Parallel computing

GPU programming is nothing else than an application of parallel computing, that is a type of computing architecture in which several processors simultaneously execute multiple smaller computation tasks, broken down from an overall larger complex problem.

The primary goal of parallel computing is to increase available computation power for faster application processing and problem-solving.

The popularization and evolution of parallel computing in the 21^{st} century came in response to processor frequency scaling hitting the power wall. In fact, the higher the frequency, the greater the amount of power used in a processor, and scaling the processor frequency is no longer feasible after a certain point. Therefore, programmers and manufacturers began designing parallel system software and producing power-efficient processors with multiple cores in order to address the issue of power consumption and overheating central processing units [43].

The importance of parallel computing continues to grow with the increasing usage of multicore processors and GPUs. GPUs work together with CPUs to increase the throughput of data and the number of concurrent calculations within an application. Using the power of parallelism, a GPU can complete more work than a CPU in a given amount of time [64].

There exist several standards to take benefit from specific hardware from different platforms. The two main interfaces for GPU programming are `CUDA` and `OpenCL`.

3.6.2 CUDA

As defined in Fred Oh's blog [11], `CUDA`, which stands for **C**ompute **U**nified **D**evice **A**rchitecture, is a parallel computing platform and programming model similar to the C language created by NVIDIA. `CUDA` helps developers speed up their applications by harnessing the power of GPU accelerators. As `CUDA` is a proprietary API, it is only supported on NVIDIA's GPUs, which are based on Tesla Architecture [57]. As a result, since both `CUDA` and NVIDIA's GPUs are developed by the same company, `CUDA` properly matches the GPU computing characteristics, and thus offers access to features and great performance [43].

The `CUDA` programming paradigm is a combination of both serial and parallel executions and contains a special C function called the kernel, which is basically a C code that is executed concurrently on a graphics card, on a fixed number of threads.

3.6.3 OpenCL

`OpenCL` an acronym for **O**pen **C**omputing **L**anguage is a cross-platform, open, royalty-free standard for parallel programming of diverse accelerators found in supercomputers, cloud servers, personal computers, mobile devices, and embedded platforms [46]. It was launched by Apple and the Khronos group as a way to provide a benchmark for heterogeneous computing that was not restricted to only NVIDIA GPUs. This last feature is probably the most recognized difference between the `CUDA` and `OpenCL`, as `CUDA` only runs on NVIDIA GPUs while `OpenCL` is an open

industry standard and runs on NVIDIA, **AMD**, Intel, and other hardware devices. This portable language is used to design programs that are general enough to run on considerably different architectures while still being adaptable enough to allow each hardware platform to achieve high performance.

Unlike the `CUDA` kernel, an `OpenCL` kernel can be compiled at runtime. Consequently, it increases the execution time of an `OpenCL` program. However, on the other side, this just-in-time compile could allow the compiler to generate code that will make better use of the target GPU.

Besides, one can also note that, as opposed to `CUDA`, `OpenCL` provides CPU fallback, which means that developers are not forced to put if-statements to distinguish between the presence of a GPU device at runtime or its absence.

For the implementation of this task, we rather opted for `CUDA`. Although `OpenCL` offers the greatest portability, `OpenCL` is generally not supported on NVIDIA modules (not supported on AGX module). Furthermore, since all the machines used by Deltatec have NVIDIA GPUs, the portability restriction of `CUDA` was not a problem either. One should also note that, in terms of performance, when a machine supports both `CUDA` and `OpenCL`, `CUDA` performance is better than `OpenCL` performance [57].

3.6.4 Acceleration of data interleaving from Linux decoder to PCIe card

As analyzed in section 3.4, visualizing the decoded data during decoding requires some data processing. In this project, we used a Deltatec PCIe card. However, the data output by the Jetson Xavier AGX hardware was not directly compatible with the DELTA-3G-elp-key 11 card. The processing to make the data compatible was done on the CPU. However, this task is completely parallelizable, and it is legitimate to wonder if the GPU programming would not allow accelerating the processing. Therefore, in this subsection, we will focus on the implementation of the data interleaving from NVIDIA hardware to the Deltatec card using the GPU.

3.6.4.1 Implementation

The implementation of this task is inspired by CUDA tutorial [48].
In order to take advantage of parallelism in programming, one needs to find a way to communicate an instruction sequence that is executable by the GPU. In the context of `CUDA`, this type of function is called a kernel and is characterized by the `__global__` specification, which allows the compiler to specify that this piece of code is executable on the GPU and can be called from the CPU. In the `CUDA` paradigm, code that runs on the GPU is often called device code, while code that runs on the CPU is host code.

To compute on the GPU, it is necessary to give memory access to GPU. Unified Memory in `CUDA` makes this easy by providing a single memory space accessible by all GPUs and CPUs in the working system (see Figure 3.17).

Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. This avoids constantly deep copy data from one device type to the other. The key is that the system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.

It is important to note that as `CUDA` kernel launches do not block the calling CPU thread, the CPU has to wait until the kernel is done before it accesses the results, which is done by only calling a function from the `CUDA API`: `cudaDeviceSynchronize()`.

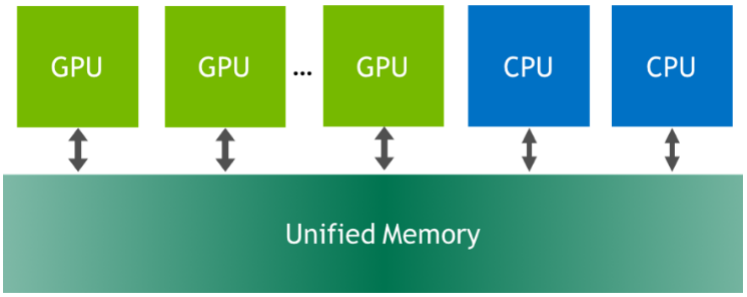


Figure 3.17: CUDA unified memory

Source: `https://developer.nvidia.com/blog/unified-memory-in-cuda-6/`

When calling a `CUDA` kernel, there are two parameters to take into account: the number of threads per block and the number of thread blocks. Both together tell how many parallel threads to use for the launch on the GPU. These parameters are passed between `<<<...>>>` after the kernel name as depicted in the following code line.

```
kernel_name<<<nbBlocks, nbThreadsPerBlock>>>(...);
```

`CUDA` GPUs run kernels using blocks of threads that are a multiple of 32. In order to properly split the computation over all the threads, `CUDA` gives access to two important variables from the device (GPU), namely, the index of the current thread within its block, `threadIdx.x`, and the number of threads in the block, `blockDim.x`.
`CUDA` GPUs have many parallel processors grouped into **S**treaming **M**ultiprocessors or **SM**s. Each SM can run multiple concurrent thread blocks. Consequently, in `CUDA`, to take full advantage of all these threads, it is recommended to spread the computation over multiple thread blocks, which is the second parameter passed when calling a `CUDA` kernel. Together, the blocks of parallel threads make up what is known as the grid. Additionally to `threadIdx.x` and `blockDim.x`, `CUDA` provides two more variables `gridDim.x`, which is the number of blocks in the grid, and `blockIdx.x`, which is the index of the current thread block in the grid. The relations between these variables can be found in Figure 3.18.

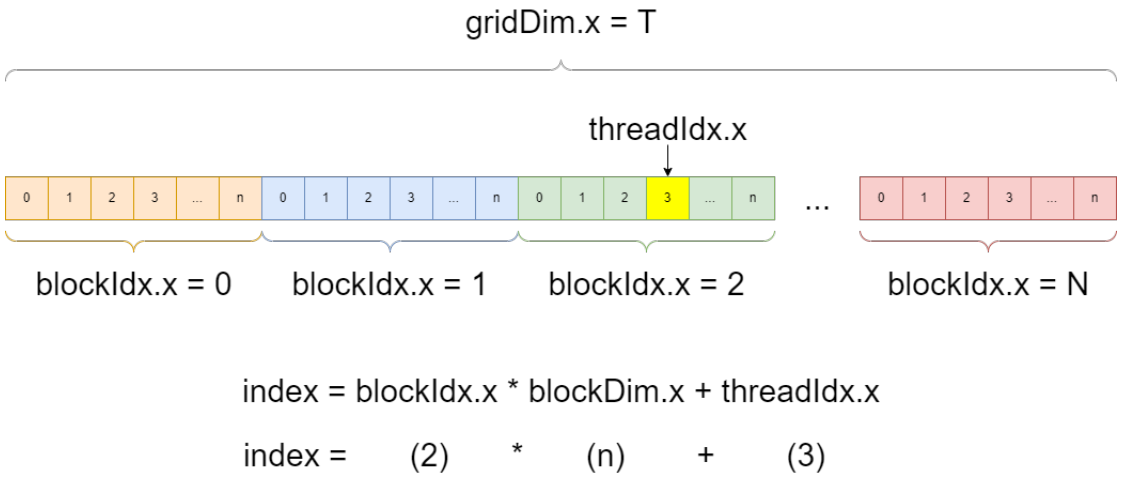


Figure 3.18: CUDA variables ralationship

Source: `https://developer.nvidia.com/blog/even-easier-introduction-cuda/`

As part of the data transformation from the NVIDIA hardware to the PCIe card, the

implementation consisted of allocating space for the correctly filled planes related to Y, U, and V for the GPU and interleaving the data on GPU. The kernel responsible for this task was called from the `HDMI` class, which outputs the stream on HDMI.

3.6.4.2 Results

In this subsection, let us study a performance comparison between CPU and `CUDA` data interleaving. To do so, we ran the program for increasing array sizes. Indeed, as a reminder, the goal of data interleaving was to transform three planes (Y, U, and V) into a single interleaved plane. Since the chroma subsampling used was 4:2:2, the Y plane is twice as large as the other two planes. It is important to note that the size of the array considered in our study is the Y plane size. The result of this study can be seen in the the Figure below.

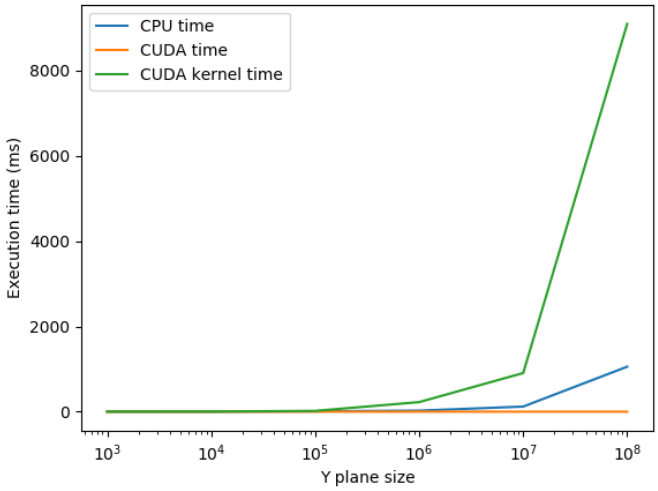


Figure 3.19: Performance comparison between CPU and CUDA data interleaving

Surprisingly, one can see the inefficiency of the `CUDA` solution. Indeed, the `CUDA` solution takes more time, whatever the size of the Y plane. Even worse, the gap between the execution times of the `CUDA` solution and the CPU-based one grows with the Y plane size.
In the case of frame transmission, which is of particular interest in this thesis, one must consider a Y plane of about 2 million pixels[20]. In this case, `CUDA` gives a data interleaving time of 1.07 seconds versus 40.88 ms for interleaving per CPU. This means that the `CUDA` solution is about 26 times slower than the CPU version.
However, if one takes a closer look, thanks to the orange curve in Figure 3.19, one can see that the `CUDA` solution is more efficient in terms of computation, *i.e.*, when we consider the execution time of the kernel, the heart of the processing. This observation leads us to suspect the presence of such a large overhead that the `CUDA` solution becomes slower than the CPU solution. In order to detect what is at the origin of this issue, one can use a powerful tool called `NVIDIA Nsight Systems`.

3.6.4.2.1 NVIDIA Nsight Systems `NVIDIA Nsight Systems` is a system-wide performance analysis tool designed to visualize the application's algorithms. It helps identify performance bottlenecks in applications from a system-level view, including multi-core CPU analysis, thread state analysis, call-stack analysis, and NVIDIA `CUDA` workload analysis.

When profiling the `CUDA` acceleration of data interleaving used in this project (see Figure 3.20), one can obviously notice the overhead related to memory management. As a matter of fact,

[20]Indeed, a 1080p frame is $1080 \times 1920 = 2073600$ pixels

`cudaMalloc` represents almost half of the total execution time. Even though the computation per se is more efficient (kernel time versus CPU time), the overhead is so huge that the overall performance is poorer compared to the CPU-based solution.

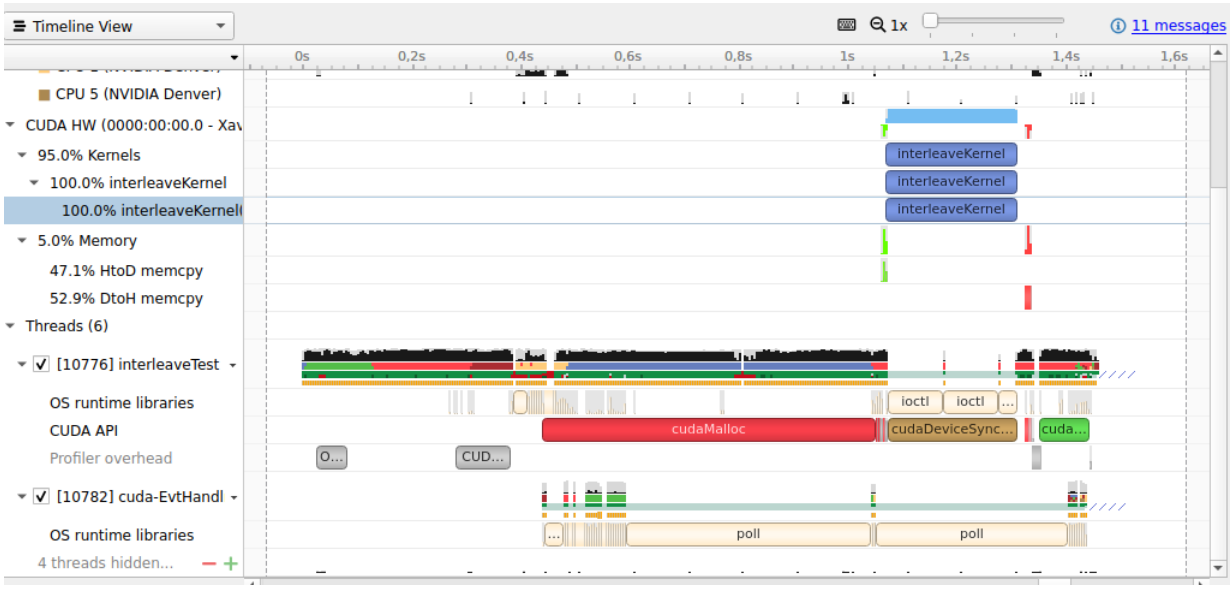


Figure 3.20: CUDA data interleaving profiling

3.6.4.3 CUDA limitations

Although there is great potential in parallelization, there are significant overheads to consider in practice. Indeed, processes such as memory management from host to device and from device to host or the initialization of `CUDA` are performance sensitive. This task is proof that despite parallelism, performance can decrease compared to a CPU-based solution.

Besides, one must point out that `CUDA` optimization of data interleaving was not the focus of this thesis. Hence, the `CUDA` implementation was not optimized. Therefore, it may be possible to optimize the `CUDA` solution so that the performance would be better than a CPU solution. Moreover, for the use made of this data processing (display frames as they are decoded on HDMI port), the tests carried out during this project never affirmed that it was a bottleneck. Nevertheless, a possible optimization path would have been to improve the data transfer. For instance, one could use the GPU data directly instead of calling the `CUDA` function: `cudaMemcpy`. Indeed, the data did not need to be stored in the host anymore as soon as it had been processed.

3.6.5 Discrete Cosine Transform

In the context of decoding acceleration using `CUDA`, we focused on specific parts of decoding, such as the DCT process. Indeed, each frame is split into several macroblocks, each one processed independently from the others, allowing a rather direct parallelization.

In order to work incrementally during this thesis, we thought it wise to start by implementing a performance comparison between the CPU DCT version and the `CUDA` DCT one on a single frame. In this subsection, we will focus on the structure of this implementation as well as the results obtained.

3.6.5.1 Implementation

This application consists in loading an image in **BMP** format[21] in black and white to apply the discrete cosine transform on it. First, the program checks that the size of the image is compatible with the application of the DCT algorithm, *i.e.*, it checks whether the width, as well as the height of the image, are divisible by the dimension of a macroblock (here, eight). If this is not the case, the application takes care of resizing the image appropriately downwards. Then, the program applies the DCT on the image and quantizes the DCT matrix before doing the inverse process, which is the IDCT to obtain the compressed image. The program performs this sequence of operations a number of times via CPU and via `CUDA`. Thus, the computation time for both methods is measured and averaged to obtain a reliable measurement. Finally, the program saves the result in a black and white BMP image to ensure the process accuracy.
One must note that, although the data is likely to be greatly compressed due to the large number of zeros introduced by the DCT process, this task does not compress the data since all values are stored in BMP format.

The code structure is illustrated in the following Figure 3.21.

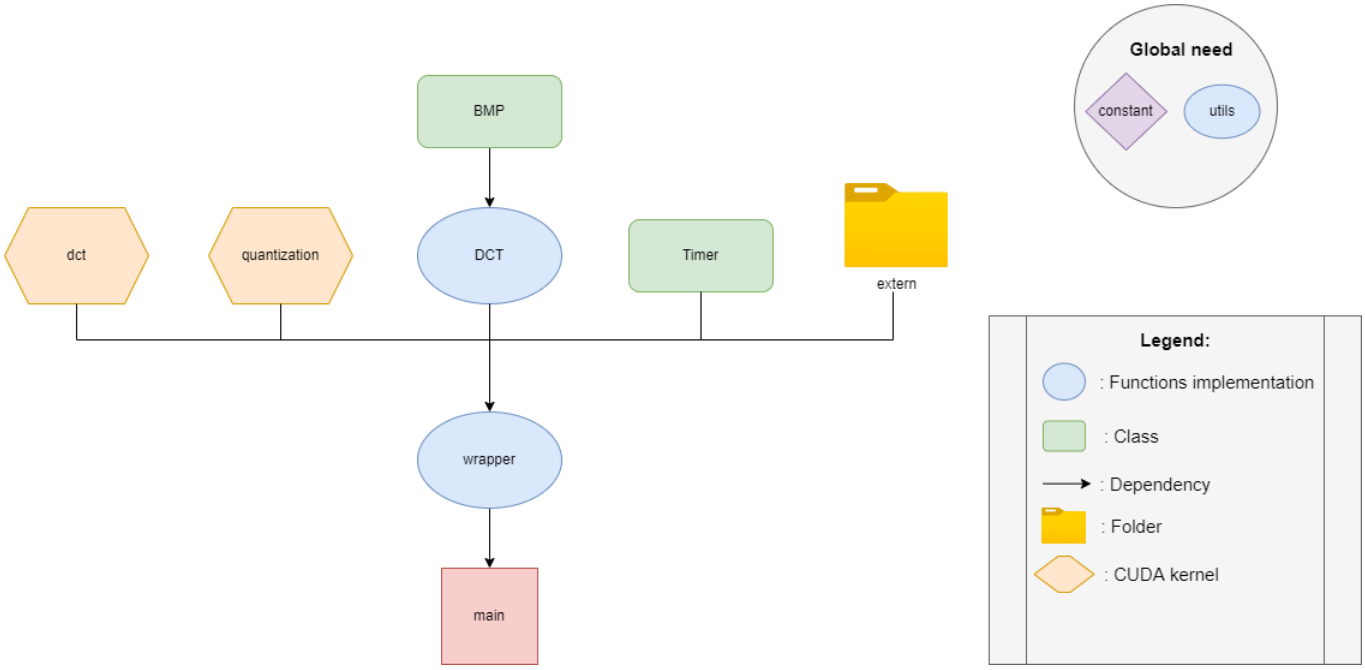


Figure 3.21: Code structure

- **constant**: This is a header file containing the main constants used in this application.
- **utils**: This is a header file that encapsulates all the functions that are not linked to a specific class or header file but are still useful throughout the application.
- **dct**: This is a set of `CUDA` kernels related to the discrete cosine transform. It implements the DCT algorithm along with its inverse, IDCT.
- **quantization**: This is a `CUDA` kernel that implements the quantization of a DCT matrix.
- **BMP**: This class represents an image in BMP format. It implements the management of such a format, *i.e.*, the loading, the saving, *etc.*
- **DCT**: This is a header file that implements the discrete cosine transform algorithm, as well as the quantization, in a CPU fashion.

[21]The BMP format is one of the simplest format that stores pixels as an array of points and manages colors either in true color or through an indexed palette.

- **Timer**: This class represents a timer. It notably allows to start, stop, reset a timer.
- **extern**: This folder contains all the external implementations, namely some `CUDA` helper functions.
- **wrapper**: This is a header file that encapsulates mainly the data management related to the DCT processing implemented in the DCT header and dct kernel.
- **main**: This is the entry point of the application.

3.6.5.2 Results

As explained earlier in section 2.3.1, it is in the quantization that the compression is irreversible. Thus, there exists a slight loss of quality, as can be noticed in the following set of Figures 3.22.

(a) Original image (b) CPU DCT image (c) CUDA DCT image

Figure 3.22: Discrete Cosine Transform results comparison

Obviously, as the algorithm itself does not change from the CPU version to the `CUDA` version, the result in terms of the output image is the same. The only difference lies in the execution time required.

`CUDA` provides excellent results since the GPU acceleration gives a solution that is more than 100 times faster compared to the CPU version. Indeed, in the case of a 512×512 image, the average DCT time (over 100 executions) is 12.91 ms for the CPU-based solution while it is only 0.11 ms for the `CUDA`-based solution. This represents an acceleration ratio of exactly 117.36 times in favor of the `CUDA` DCT.
It is interesting to note that the CPU version is single-threaded.

3.6.5.2.1 Resolution influence To find out whether such acceleration was independent of the resolution considered or not, we decided to also compare the execution times, as well as the acceleration ratios, for different resolutions images. The results of this experiment can be found in the following Table 3.7.

	Resolution				
	4K	**2K**	**1080p**	**720p**	**480p**
CPU time (ms)	1197.21	300.72	89.79	39.7	17.23
CUDA time (ms)	22.21	5.55	1.45	2	1
Acceleration ratio	53,9	54.18	61.92	19.85	17.23

Table 3.7: CUDA vs CPU DCT *w.r.t* image resolution

In order to carry out this test in the most relevant way, we considered the same image with different resolutions. Indeed, the image content varies the ease with which the image is

compressed and hence varies the decoding time.

From Table 3.7 above, one can notice that whatever the image resolution used, the DCT algorithm accelerated via `CUDA` is always much faster than the CPU solution (at least 17 times faster). Moreover, one logically observes that below a certain resolution, in this case, below 1080p, the `CUDA` decoding time no longer decreases significantly, which implies a decrease in acceleration for a decreasing resolution. This is mainly because using `CUDA` involves a certain amount of overhead (memory allocation, data transfer management, *etc.*) which inevitably increases the execution time. Moreover, the fewer data to process, the more this overhead takes up a large part of the total execution time.
However, beyond 1080p, given the tests performed, it is difficult to come to an unequivocal conclusion concerning the influence of the resolution on the speed-up. Nevertheless, one may believe that the acceleration is independent of the image resolution.

3.6.5.2.2 Standard deviation of execution times An important feature to consider that may influence the study is that the standard deviation in execution times was generally large in the studied cases (see Table 3.8). This standard deviation can be justified by the fact that the execution times studied were short, in the order of seconds at most, and even in the order of milliseconds. At this level, OS scheduling has a strong influence. This is why we can find that the variance decreases proportionally when the execution time increases.

Standard deviation	**Resolution**				
	4K	**2K**	**1080p**	**720p**	**480p**
CPU (ms)	21.48	13.2	7.66	4.87	1
CPU (%)	1.79	4.39	8.53	12.27	5.8
CUDA (ms)	0.27	4.64	1.2	0	0
CUDA (%)	1.22	83.6	82.76	0	0

Table 3.8: Standard deviation in execution time

3.6.6 CUDA acceleration integration in a complete decoding solution

Having implemented and tested the efficiency of `CUDA` in accelerating the DCT process, the aim was to integrate this into the workflow used by `FFmpeg` and then to compare the results of the three types of decoder implemented in this thesis, namely a hardware-accelerated decoder, a fully CPU-based decoder, and a `CUDA`-accelerated decoder, the last decoder being the best trade-off between portability and performance.

3.6.6.1 Implementation

The general structure of the class diagram is illustrated in the following Figure 3.23.

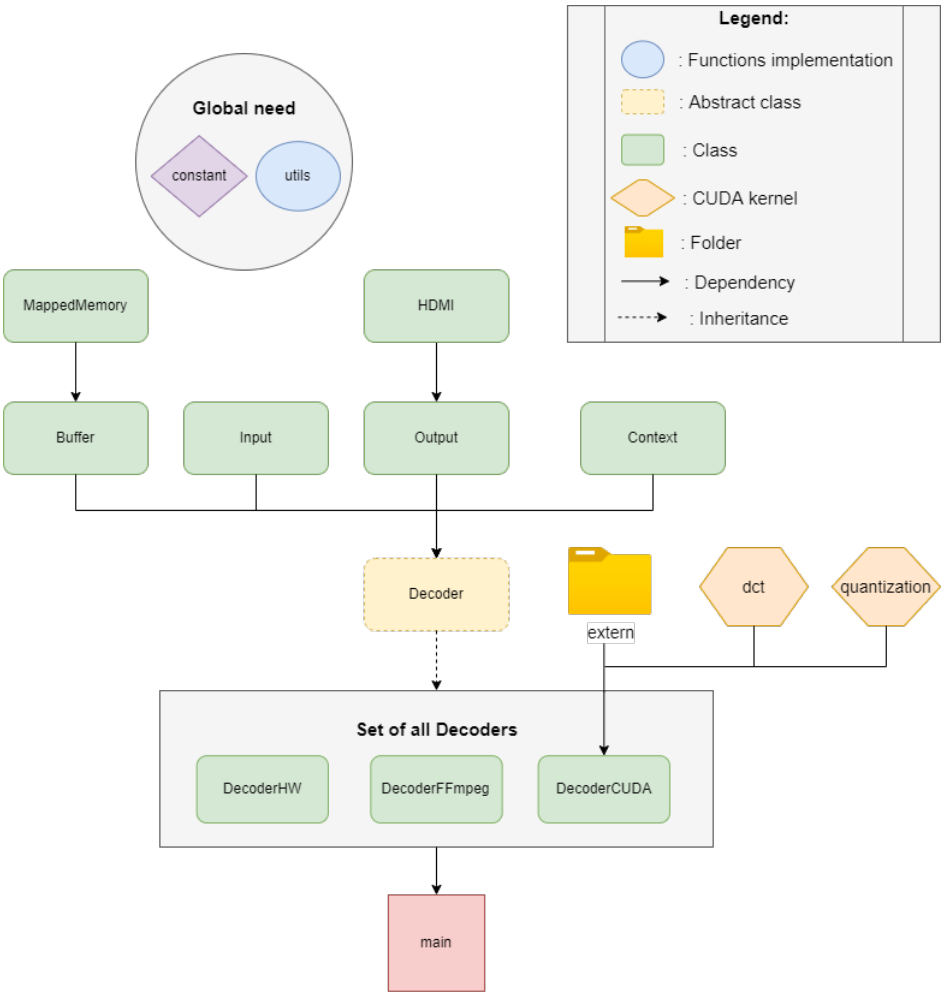


Figure 3.23: Code structure

The integration of the new `CUDA`-based decoder uses the inheritance structure between decoders described in section 3.5. As mentioned earlier, `CUDA` relies on functions called kernels which are separated from the C++ code for clarity. Therefore, the latter decoder has more links that are specific to `CUDA` than the other decoders in the diagram. In addition, as mentioned in subsection 3.6.5, to implement the DCT algorithm acceleration, we used a number of external functions (`CUDA` helpers) which are also used by the `CUDA` decoder and contained in the `extern` folder.

Regarding the workflow, it is fully similar to the one described in Figure 3.14 of section 3.5. The only difference lies in the packet decoding stage where the decoder makes use of the `CUDA` accelerated DCT algorithm.

3.6.6.2 Results

With the limited time available, we were unfortunately unable to achieve a functional integration of the GPU-accelerated DCT algorithm into the `FFmpeg` decoding workflow. Therefore, the comparison of the three types of decoding: hardware, CPU, and `CUDA`, is not possible in the context of this thesis.

3.7 NVIDIA Jetson Xavier AGX : NDI stream

The last implementation part discussed in this thesis consisted of adapting the decoding application to handle the decoding of streams coming from the network. As detailed at length in section 2.5, the audiovisual world has been in transition to video over IP for some time, of which the NDI protocol is the best known and most widespread. Consequently, this is the protocol on which the application is based in order to decode streams and not only files.

3.7.1 NDI SDK

The NDI SDK is a royalty-free SDK and provides the tools and resources for developers and manufacturers to easily add native NDI support to their video products. The SDK enables systems to find, send, and/or receive video streams over IP, with an encoding algorithm independent of resolution and frame rate supporting up to 4K (and beyond).

3.7.2 Implementation

The implementation of this feature did not involve major changes in the class diagram presented so far (see Figure 3.24).

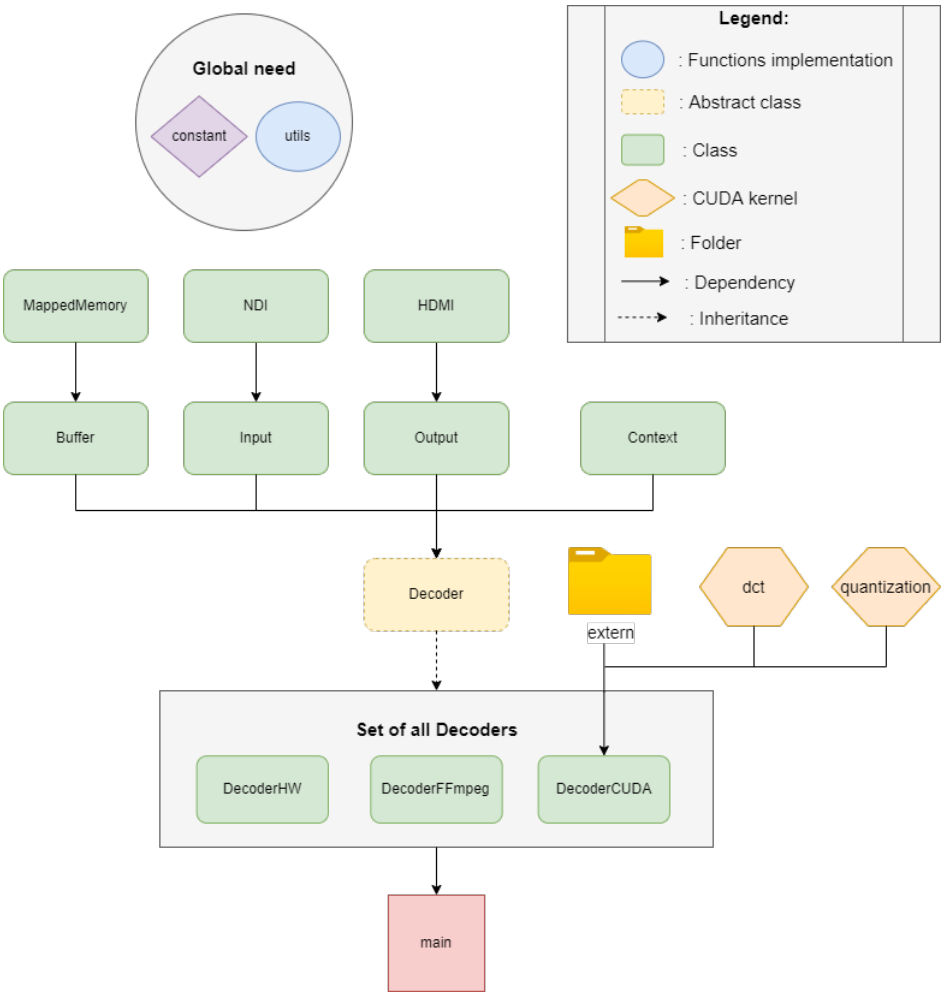


Figure 3.24: Code structure

As a matter of fact, the notable change compared to diagram 3.23 in the previous section is the addition of the `NDI` class, which is the class representing and implementing the management of NDI streams. This class is logically used by the `Input` class, which, consequently, reads data from either an input file or an NDI stream.

When the application deals with an NDI stream, the `NDI` class takes care of finding all the sources available on the LAN network and connects to the source specified by the user. Thus, every video frame is captured by the `NDI` class and forwarded to the `Input` class. Finally, the `Input` class is responsible for filling one of the available hardware buffers, when it comes to the hardware decoding or transforming the frame defined by the NDI SDK into a packet defined by the `FFmpeg` library for the `FFmpeg`-based decoding. From this state, the decoding process is exactly the same as previously described.

Chapter 4

Retrospective analysis

Every project has its good and bad points. However, on the whole, we are proud of the work done and the amount of new material learned during this project. Nevertheless, one must be able to evaluate in detail what went well and what went less well in this thesis.

The organization of the work, as well as the division of the objectives into smaller ones, was a major strength in the realization of this thesis. Indeed, it allowed us to be on the same wavelength with Deltatec while progressing in an incremental way.

Another feature that was taken advantage of during this project was the extensive testing phase (see next section). This also contributed to the incremental progress in the project, and to the development of a quality decoding application.

On the other hand, as nothing is perfect, the work provided has limitations, which will be discussed in detail in section 4.2, and that result mainly from a choice of priorities on such and such matters. Indeed, given the time available, it was sometimes necessary to choose which task to tackle in order to study what seemed to us to be the most interesting and related to the subject dealt with here at the potential expense of another one.
Because to choose is to forsake.

4.1 Testing

As mentioned earlier, testing played a major role in the design of this project, as it is notably thanks to it that we were able to build the decoding application incrementally. Indeed, almost every new feature was first imagined and developed, then tested in a blank project, and finally integrated into the main application. All the tests carried out during this thesis can be found in the `test` folder of the project.

One can distinguish two main types of tests. On the one hand, the so-called unit tests consisted of testing a rather small functionality, such as a function or a class, in an annex project. On the other hand, the so-called integration tests consisted of assembling several independently tested functionalities into a larger application that was annexed to the main application. In the context of this project, the deployment of functional tests did not seem to us to be a priority since the functionalities were tested upstream and we rather opted for verification of the results by means of tools and visually (the analysis of video streams lends itself well to this last type of verification). Indeed, to visually test the correctness of the decoded data, the result was either stored in a file and then checked by a free software called `yuvplayer` allowing to display the result or directly displayed on the HDMI output via the Deltatec PCIe card while decoding the stream.

In the rest of this section, we will present the structure and implementation details of some unit and integration tests.

4.1.1 Discrete Cosine Transform

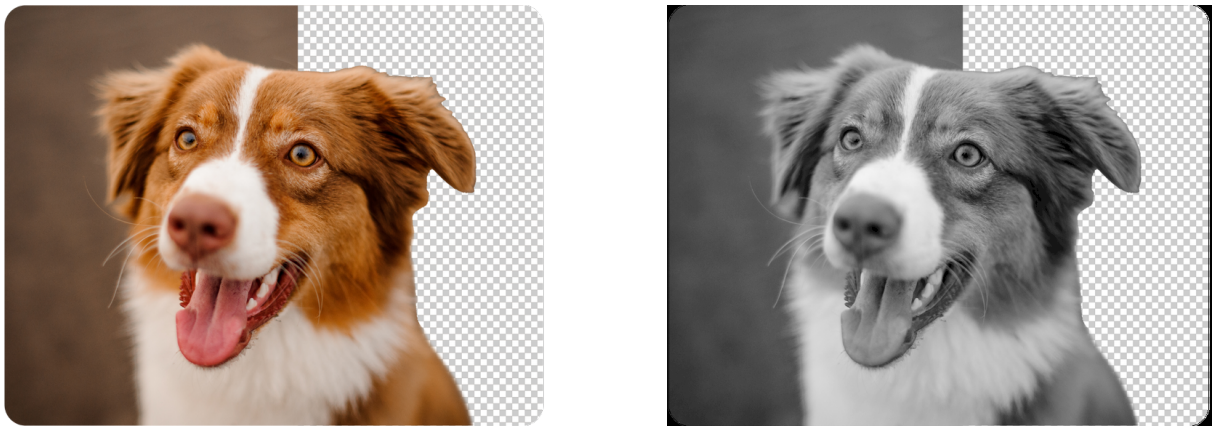
This test consisted in loading an image in memory, cutting this last one in fixed size macroblocks, applying the DCT algorithm, and finally applying the inverse process, IDCT to reobtain the initial image. This test, based on CPU programming only, was also used as a base block to establish the acceleration of the DCT process via `CUDA` detailed in section 3.6.5. However, it is important to note that this task had a pedagogical purpose to become familiar with DCT and was therefore not intended to be optimal.

One can therefore distinguish three main steps in this integration test: the management of the loading and saving of the image, the division of the image into macroblocks, and the application of the discrete cosine transform. Each of these steps was the target of a unit test. In order to carry out this integration test, we used the famous `OpenCV`[1] library, which allowed us to considerably simplify certain issues such as the image format management.

4.1.1.1 Image read and write

The first step was to load a color image into a grayscale image. The transformation of the image into black and white was necessary to avoid having to apply DCT three times, once for each color component. Once again, the accuracy of this operation was verified visually by saving the result in another image. The following Figure 4.1 shows the result of this first unit test.

[1] `OpenCV` is an open-source computer vision and machine learning software C++ library which implements more than 2000 optimized algorithms.



(a) Original image (b) Grayscale image

Figure 4.1: Color to grayscale image

This unit test implementation can be found in the file `cpuDCT/src/grayscale.cpp`.

4.1.1.2 Macroblock image division

Once the grayscale image loaded, one had to find a way to cut the original image into a multitude of macroblocks. In order to easily and visually check the process, we took advantage of the possibilities offered by `OpenCV`. Indeed, the test surrounds in red each macroblock in the image (see Figure 4.2 below).

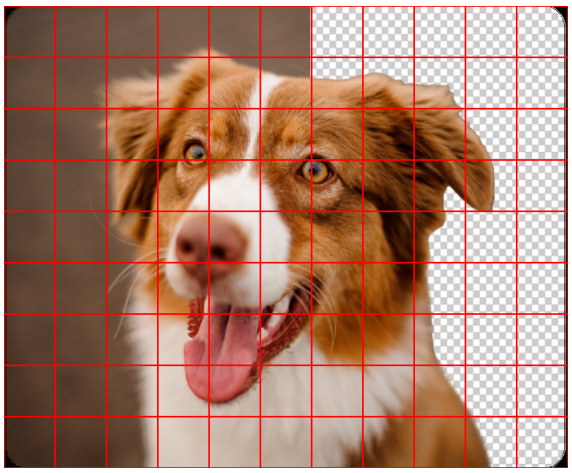


Figure 4.2: Macroblock division

The file implementing this unit test is located at: `cpuDCT/src/imageBlock.cpp`.
We made the implementation choice to redefine the dimensions of the image downwards when they were not divisible by the image dimensions. In the case illustrated above, the initial dimensions of the image were 579 $\times$ 750 pixels. Hence, the image is resized by the program to 576 $\times$ 704 so that each dimension of the image is a multiple of the block size (in the above example, the macroblocks are of size 64×64). Furthermore, it should be noted that the program also allows rectangular blocks.

4.1.1.3 DCT algorithm

The last unit test (`cpuDCT/src/dct.cpp`) consisted in applying the DCT algorithm on a fixed size matrix. The result was validated by the fact that the matrix underwent the DCT algorithm followed by the inverse IDCT algorithm. Thus, if the matrix was unchanged after these two functions, this proved to a large extent the accuracy of the procedure. To extinguish any doubts,

we cross-checked the results obtained with the implementation of the DCT algorithm in the Python package: `scipy`.

Finally, the integration test (`cpuDCT/src/cpuDCT.cpp`) successfully combines these three functionalities. The complex part of the integration was the reconstitution of the original image from the processed macroblocks.

4.1.2 Memory mapping

For this project, it was necessary to use `mmap`, which is a **UNIX system call** to map files or devices into memory when dealing with `Video4Linux`, for operations such as I/O.

To make it short and simple, in computing, what we call memory is the computer's workspace, also called RAM. This workspace allows for very fast operations, both in reading and writing. However, this space is expensive and therefore reduced compared to the storage space (hard disk, **SSD**, *etc.*). Consequently, the computer fills its memory with data that needs to be processed immediately and empties its memory (potentially saving the data in the storage space first) of data that no longer needs to be processed. This is where the system call[2] `mmap` comes in to perform the memory filling function. For such an operation to be safe in the long term, it is necessary to use `unmap`, which is the UNIX system call that takes care of deallocating the obsolete resources stored in RAM. In other words, `unmap` performs the RAM flush operation [27].

One of the key concepts of the C++ programming language is the concept of RAII. **RAII**, which stands for **R**esource **A**cquisition **I**s **I**nitialization (this does not reflect in an obvious way, from its name, the basic principles of the concept), is a programming idiom that recommends to

- Encapsulate a resource into a class, whose constructor usually (but not necessarily) acquires the resource, and its destructor always releases it.
- Use the resource via a local instance of the class.[3]

By respecting these two practices, this idiom ensures that the resource is automatically freed when the object gets out of scope. In other words, this guarantees that whatever happens while the resource is in use (normal return, destruction of the containing object, or an exception thrown), it will eventually get freed. Apart from being a safe way to deal with resources, it also makes the code cleaner as one does not need to mix error handling code with the main functionality.

As a result, it was deemed judicious to implement a class allowing to manage the memory mapping according to the RAII concept. Thus, after a certain amount of data has been mapped in memory, the destructor of such a class (named here `MappedMemory`), called automatically as soon as the object leaves the scope, takes care of calling `unmap` without the user should worry about it.

The unit test, located in the `mmap/mmap.cpp` file, allows testing the implementation of this class.
Concretely, it loads the content of a file for specified data size and file descriptor, then displays the content pointed by the data pointer of the `MappedMemory` class. By comparing the content of the file given as input with the one displayed on the standard output, one can confirm the correctness of the implementation of the `MappedMemory` class.

[2] A system call is nothing more than a function allowing the operating system to interact with the machine's hardware.

[3] *Local* refers to a local variable, or a nonstatic member variable of a class. In the latter case the member variable is initialized and destroyed with its owner object.

4.2 Limitations

In this section, we will review the main features of the decoding application and cite the limitations corresponding to each one. In other words, we will mainly describe the limitations of the different types of decoding.

As for the hardware decoding solution, the precise management of the number of frames to be decoded is not optimal. Indeed, there exists a synchronization problem between the reading thread and the decoding thread. The application as it stands stops when the reading of the number of frames specified by the user is finished, without waiting for the end of the decoding. Of course, there should be a working condition variable allowing the reading thread (which has always finished its task before the decoding thread) to wait until the decoding thread finishes.

Concerning the real-time decoding viewer, the application outputs the processed video on HDMI with a frame rate independent of the default frame rate of the video. That is to say that, whatever the video, the HDMI works with a rate of 60 fps. Therefore, for videos with a different rate, the application speeds up or slows down the video. There are two possible solutions to improve this: either the application passes the actual rate of the video to the PCIe card, and the latter then adjusts the frame rate on the HDMI output[4], or the application repeats or removes certain frames to achieve the 60 fps rate[5].

Regarding the integration of `CUDA` acceleration into the decoding application, as mentioned above, this task could not be fully completed in the time available.

Finally, as far as NDI stream decoding is concerned, only hardware decoding works correctly with the NDI stream. Indeed, as for decoding using `FFmpeg`, the transformation of the frame representation according to the NDI SDK into a frame representation according to the `FFmpeg` library, as well as the constant updating of the `FFmpeg` context, are not something trivial to implement. Indeed, the reading of the NDI stream relies on the NDI SDK, whereas `FFmpeg` builds and updates the context while reading the stream. However, the read function of the `FFmpeg` library involves many other calls whose usefulness in the context we are interested in is difficult to perceive.

[4] This requires further manipulation of the `VideoMaster SDK`.

[5] A similar operation called Three-two pull down is applied to convert 24 frames per second video into 29.97 frames per second video used in the context of TV production or other [67].

Chapter 5

Conclusion

The mature world of audiovisuals is a domain that requires both technical and complex knowledge to meet the market requirements. Indeed, the amount of raw data associated with a video is too large to transit through the Internet network and serve millions of users at the same time without congestion. The solution to such a challenge is compression. Indeed, as discussed in more detail in this thesis, video compression techniques take advantage of two major axes: spatial coding and temporal coding. On the one hand, spatial coding consists of compressing each video frame as much as possible. First of all, chroma subsampling takes advantage of the human eye's sensitivity to perceive color variations less well than gray level variations, by reducing color information. Moreover, the discrete cosine transform, which takes advantage of the human eye's sensitivity to perceive fine details (low contrast and high frequency) less well than large details, eliminates the transmission of imperceptible details by the auditor. Finally, intra-prediction aims to reduce data storage by storing mathematical functions to estimate pixel values for a certain region rather than the pixel values themselves. On the other hand, temporal coding consists of taking advantage of the redundancy between frames to compress the data. Thus, block motion estimation and compensation allow obtaining the next frame by transmitting only the information necessary to reconstruct it from the previous one. Finally, frame differencing makes it possible to correct the errors introduced by the two previous techniques when they are used alone. In addition, video production, which was, until recently, restricted to a professional world, has been democratized with the appearance of video over IP, more precisely with the NDI protocol. Indeed, NDI dematerializes the video production that used to require SDI cables and video capture cards and, at the same time, widens the video production possibilities.

This thesis allowed us to highlight the techniques of decoding multimedia streams in a very particular context, that is the embedded world. Nowadays, embedded systems are more and more used for maintenance ease, performance, and cost-efficiency, they provide. Therefore, we took in hand a development kit to program a very powerful embedded module, named NVIDIA Jetson Xavier AGX so that it can be used as a broadcast solution. A solution using the specific decoding HW was thus implemented. This solution is the most powerful, with impressive results that allow for decoding and viewing in real-time 4K resolution contents. In addition, a CPU solution was developed, that allows the decoding application to be extended to other platforms without being forced to use a specific HW. We saw that the price of such portability was not negligible since decoding times were at least doubled in the case of 4K content. The trade-off between portability and performance is best achieved by using GPU acceleration. Indeed, by accelerating parts of the decoding process, such as the discrete cosine transform, it is possible to increase performance compared to the CPU solution without affecting portability since the only constraint linked to the use of CUDA is to have NVIDIA GPUs. However, this constraint is not very restrictive in the sense that the majority of GPUs are of the NVIDIA type as this company is the market leader. We could see through this study that the DCT process was accelerated more than 50 times thanks to CUDA, independently of the high resolutions considered (1080p and more), which thus opens up perspectives of more powerful decoding using GPU programming.

5.1 Work prospects

As explained in this thesis, the world of embedded systems is expanding considerably. Furthermore, the mature but omnipresent audiovisual industry is constantly being updated and improved. Therefore, this thesis, of course, opens the door to several upcoming research projects.
To name a few, in the future, it would be interesting to study and quantify the performance of adding video overlays (subtitles, logos, *etc.*) via the NVIDIA Jetson Xavier AGX module using, for instance, hardware, CPU, or `CUDA`.
Furthermore, studying the reverse process of decoding, *i.e.*, encoding, in an analogous way to this thesis is also an interesting future work prospect.
Finally, although it is less innovative, improving or even extending the solutions developed so far concerning the application limitations, discussed in section 4.2, is no less valuable.

5.2 Final words

This Master thesis has allowed me to learn lots of new matters and acquire many new skills. Indeed, thanks to this long and extensive project, the field of multimedia, more precisely the compression of multimedia streams is a subject that I master. Moreover, through this project, I had the chance to manipulate state-of-the-art hardware in the development of embedded systems, a world in which I learned from scratch thanks to this thesis. Moreover, I had the opportunity to carry out this thesis in collaboration with Deltatec, a company that hosts a large number of qualified engineers with whom I have grown enormously, especially in the organization of large projects.
Finally, I am proud to have done quality work in a subject as complex as the world of video streaming on embedded systems.

Bibliography

[1] Steve Heath. "1 - What is an embedded system?" In: *Embedded Systems Design (Second Edition)*. Ed. by Steve Heath. Second Edition. Oxford: Newnes, 2002, pp. 1–14. ISBN: 978-0-7506-5546-0.

[2] Walter Ciciora et al. "Chapter 3 - Digitally Compressed Television". In: *Modern Cable Television Technology (Second Edition)*. Ed. by Walter Ciciora et al. Second Edition. The Morgan Kaufmann Series in Networking. San Francisco: Morgan Kaufmann, 2004, pp. 71–136.

[3] Michael Igarta. "A study of MPEG-2 and H.264 video coding". Electrical and Computer Engineering. Purdue University, Dec. 2004.

[4] D. Austerberry. "4 - Video formats". In: *The Technology of Video and Audio Streaming, Second Edition*. Ed. by D. Austerberry. Second Edition. Elsevier/Focal Press, 2005, pp. 52–77.

[5] D. Austerberry. "5 - Video compression". In: *The Technology of Video and Audio Streaming, Second Edition*. Ed. by D. Austerberry. Second Edition. Elsevier/Focal Press, 2005, pp. 78–101.

[6] Jiheng Yang et al. "A Block-Matching Based Intra Frame Prediction for H.264/AVC". In: *2006 IEEE International Conference on Multimedia and Expo*. 2006, pp. 705–708.

[7] Huifang Sun, Anthony Vetro, and Jun Xin. "An overview of scalable video streaming". In: *Wireless Communications and Mobile Computing* 7.2 (2007), pp. 159–172.

[8] Anton Obukhov and Alexander Kharlamov. *Discrete Cosine Transform for 8x8 Blocks with CUDA*. Oct. 2008.

[9] Jizheng Xu and Bing Zeng. "An overview of directional transforms in image coding". In: May 2010, pp. 3036–3039.

[10] Mamoona Naveed Asghar and Mohammad Ghanbari. "MIKEY for keys management of H.264 scalable video coded layers". In: *Journal of King Saud University - Computer and Information Sciences* 24.2 (2012), pp. 107–116. ISSN: 1319-1578.

[11] Fred Oh. *What Is CUDA?* NVIDIA. Sept. 2012. URL: `https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/`.

[12] "12 - Introduction to luma and chroma". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 121–128. ISBN: 978-0-12-391926-7.

[13] "16 - Introduction-to-video-compression". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 147–162. ISBN: 978-0-12-391926-7.

[14] "45 - JPEG and motion-JPEG (M-JPEG) compression". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 491–504. ISBN: 978-0-12-391926-7.

[15] "47 - MPEG-2 video compression". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 513–536. ISBN: 978-0-12-391926-7.

[16] "48 - H.264 video compression". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 537–548. ISBN: 978-0-12-391926-7.

[17] "49 - VP8 compression". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 549–552. ISBN: 978-0-12-391926-7.

[18] "6 - Raster images in computing". In: *Digital Video and HD (Second Edition)*. Ed. by Charles Poynton. Second Edition. The Morgan Kaufmann Series in Computer Graphics. Boston: Morgan Kaufmann, 2012, pp. 65–73. ISBN: 978-0-12-391926-7.

[19] vcodexer. *Vcodex: Introduction to Video Coding*. Youtube. June 2013. URL: https://www.youtube.com/watch?v=gxefuXizO04&ab_channel=vcodexer.

[20] Nishu Singla. "Motion Detection Based on Frame Difference Method". In: *International Journal of Information & Computation Technology* 4,15 (2014), pp. 1559–1565.

[21] Paul Bourke. *NV12 yuv pixel format*. Aug. 2016. URL: http://paulbourke.net/dataformats/nv12/.

[22] Bill Dirks et al. *Part I: Video for Linux API*. Tech. rep. Madison, WI, USA: The kernel development community, 2016. URL: https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/v4l2.html.

[23] HandyAndy Tech Tips. *H.265 (HEVC) vs H.264 (AVC) Compression: Explained!* Youtube. 2017. URL: https://www.youtube.com/watch?v=Fawcboio6g4&t=151s&ab_channel=HandyAndyTechTips.

[24] Gagne Silberschatz Galvin and Mathy. *Chapter 1: Introduction*. Operating systems, INFO0940-1. Accessed: 2021–02-04. Feb. 2018.

[25] Gagne Silberschatz Galvin and Mathy. *Chapter 2: Operating system services*. Operating systems, INFO0940-1. Accessed: 2021–02-10. Feb. 2018.

[26] Gagne Silberschatz Galvin and Mathy. *Chapter 5: CPU scheduling*. Operating systems, INFO0940-1. Accessed: 2021–03-03. Mar. 2018.

[27] Gagne Silberschatz Galvin and Mathy. *Chapter 6: Main memory*. Operating systems, INFO0940-1. Accessed: 2021–04-06. Apr. 2018.

[28] NVIDIA Developer. *NVIDIA Jetson AGX Xavier Developer Kit - Introduction*. Youtube. 2019. URL: https://www.youtube.com/watch?v=XoWW5HiGHsg&ab_channel=NVIDIADeveloper.

[29] Leo Isikdogan. *How image compression works*. Youtube. 2019. URL: https://www.youtube.com/watch?v=Ba89cI9eIg8&ab_channel=LeoIsikdogan.

[30] Leo Isikdogan. *How video compression works*. Youtube. 2019. URL: https://www.youtube.com/watch?v=Ba89cI9eIg8&ab_channel=LeoIsikdogan.

[31] NVIDIA Corporation. *Jetson AGX Xavier developer kit user guide*. Tech. rep. 2019.

[32] NVIDIA Developer. *Jetson AGX Xavier and the new era of autonomous machines*. Youtube. 2020. URL: https://www.youtube.com/watch?v=dG2iNaz1ggc&ab_channel=NVIDIADeveloper.

[33] Video Tech Explained. *What are Color Spaces?* Youtube. 2020. URL: hhttps://www.youtube.com/watch?v=WLF3uqb5otM&t=89s&ab_channel=VideoTechExplained.

[34] Brad Niepceron, Ahmed Nait Sidi Moh, and Filippo Grassia. "Moving Medical Image Analysis to GPU Embedded Systems: Application to Brain Tumor Segmentation". In: *Applied Artificial Intelligence* 34 (July 2020), pp. 1–14. DOI: 10.1080/08839514.2020.1787678.

[35] David R. Bull and Fan Zhang. "Chapter 1 - Introduction". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 1–16. ISBN: 978-0-12-820353-8.

[36] David R. Bull and Fan Zhang. "Chapter 12 - Video coding standards and formats". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 435–484. ISBN: 978-0-12-820353-8.

[37] David R. Bull and Fan Zhang. "Chapter 2 - The human visual system". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 107–142. ISBN: 978-0-12-820353-8.

[38] David R. Bull and Fan Zhang. "Chapter 4 - Digital picture formats and representations". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 107–142. ISBN: 978-0-12-820353-8.

[39] David R. Bull and Fan Zhang. "Chapter 5 - Transforms for image and video coding". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 143–182. ISBN: 978-0-12-820353-8.

[40] David R. Bull and Fan Zhang. "Chapter 7 - Lossless compression methods". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 225–270. ISBN: 978-0-12-820353-8.

[41] David R. Bull and Fan Zhang. "Chapter 8 - Coding moving pictures: motion prediction". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 271–308. ISBN: 978-0-12-820353-8.

[42] David R. Bull and Fan Zhang. "Chapter 9 - The block-based hybrid video codec". In: *Intelligent Image and Video Compression (Second Edition)*. Ed. by David R. Bull and Fan Zhang. Second Edition. Oxford: Academic Press, 2021, pp. 309–333. ISBN: 978-0-12-820353-8.

[43] Pascal Fontaine. *Introduction*. Parallel programming, INFO9012-1. Accessed: 2021–02-04. 2021.

[44] NVIDIA. *Jetson Linux API Reference - Multimedia APIs*. Tech. rep. NVIDIA CORPORATION & AFFILIATES., 2021. URL: https://docs.nvidia.com/jetson/l4t-multimedia/mmapi_group.html.

[45] Paul W. Richards. *The Unofficial Guide to NDI*. Tech. rep. 2021.

[46] Khronos Group. *OpenCL Guide*. Tech. rep. Khronos Group Inc. OpenCL, 2022. URL: https://github.com/KhronosGroup/OpenCL-Guide.

[47] *Advanced Video Coding*. Wikipedia. URL: https://en.wikipedia.org/wiki/Advanced_Video_Coding.

[48] *An Even Easier Introduction to CUDA*. NVIDIA Developer. URL: https://developer.nvidia.com/blog/even-easier-introduction-cuda/.

[49] Deltatec. *VideoMaster SDK - Programming guide*.

[50] *Deltatec's page*. Deltatec. URL: https://www.deltatec.be.

[51] *Device driver*. Wikipedia. URL: https://en.wikipedia.org/wiki/Device_driver.

[52] *Graphics processing unit*. Wikipedia. URL: https://en.wikipedia.org/wiki/Graphics_processing_unit.

[53] *Hardware Acceleration*. HEAVY.AI. URL: https://www.heavy.ai/technical-glossary/hardware-acceleration.

[54] *High Efficiency Video Coding*. Wikipedia. URL: https://en.wikipedia.org/wiki/High_Efficiency_Video_Coding.

[55] *Install Jetson Software with SDK Manager*. NVIDIA. URL: https://docs.nvidia.com/sdk-manager/install-with-sdkm-jetson/index.html.

[56] *Internet of things*. Wikipedia. URL: https://en.wikipedia.org/wiki/Internet_of_things.

[57] Kamran Karimi, Neil G. Dickson, and Firas Hamze. *A Performance Comparison of CUDA and OpenCL*.

[58] *La fibre - Proximus*. Proximus. URL: https://www.proximus.be/fr/id_cl_opticalfibersolutions/entreprises-et-secteur-public/reseaux/fibre-optique/solutions-fibre.html.

[59] *Machine Learning*. IBM. URL: https://www.ibm.com/cloud/learn/machine-learning.

[60] *Machine vision*. Wikipedia. URL: https://en.wikipedia.org/wiki/Machine_vision.

[61] *MPEG-2*. Wikipedia. URL: https://en.wikipedia.org/wiki/MPEG-2.

[62] *NVIDIA*. Wikipedia. URL: https://fr.wikipedia.org/wiki/Nvidia.

[63] *Overhead (computing)*. Wikipedia. URL: https://en.wikipedia.org/wiki/Overhead_(computing).

[64] *Parallel Computing*. HEAVY.AI. URL: https://www.heavy.ai/technical-glossary/parallel-computing.

[65] *RGB color model*. Wikipedia. URL: https://en.wikipedia.org/wiki/RGB_color_model.

[66] *scp command in Linux with Examples*. GeeksforGeeks. URL: https://www.geeksforgeeks.org/scp-command-in-linux-with-examples/.

[67] *Three-two pull down*. Wikipedia. URL: https://en.wikipedia.org/wiki/Three-two_pull_down#%5C%3A~%5C%3Atext%5C%3DThree-two%5C%20pull%5C%20down%5C%20(3%5C%2Cslight%5C%20slow%5C%20down%5C%20in%5C%20speed..

[68] *Units of information*. Wikipedia. URL: https://en.wikipedia.org/wiki/Units_of_information.

[69] *UNIX Full Form*. GeeksforGeeks. URL: https://www.geeksforgeeks.org/unix-full-form/#:~:text=UNIX%5C%20was%5C%20earlier%5C%20known%5C%20to,a%5C%20variety%5C%20of%5C%20platforms(Eg..

[70] *Video coding format*. Wikipedia. URL: https://en.wikipedia.org/wiki/Video_coding_format#:~:text=A%5C%20video%5C%20coding%5C%20format%5C%20(or,a%5C%20data%5C%20file%5C%20or%5C%20bitstream)..

[71] *Virtual reality*. Wikipedia. URL: https://en.wikipedia.org/wiki/Virtual_reality.

[72] *VP9*. Wikipedia. URL: https://en.wikipedia.org/wiki/VP9.

[73] *What's the difference between NDI and NDI|HX?* AVONIC. URL: https://avonic.com/whats-the-difference-between-ndi-and-ndihx/.

[74] *YCbCr*. Wikipedia. URL: https://en.wikipedia.org/wiki/YCbCr.

[75] *YUV*. Wikipedia. URL: https://en.wikipedia.org/wiki/YUV.

[76] *YUV 420, YCbCr 422, RGB 444, c'est quoi le chroma subsampling 2?* L'atelier du câble. URL: http://www.latelierducable.com/tv-televiseur/yuv-420-ycbcr-422-rgb-444-cest-quoi-le-chroma-subsampling/.

Acronyms

AMD Advanced Micro Devices. 62

API Application Programming Interface. 42

ASP Advanced Simple Profile. 31

AVC Advanced Video Coding. 30

bpp bit per pixel. 29

CIE International Commission on Illumination. 13

CPU Central Processing Unit. 42

CUDA Compute Unified Device Architecture. 2, 61

DCT Discrete Cosine Transform. 17

DFT Discrete Fourier Transform. 20

DVD Digital Video Disc. 30

FDCT Forward Discrete Cosine Transform. 20

FIFO First In First Out. 43

fps frame per second. 29

GOP Group Of Pictures. 28

GUI Graphical User Interface. 3

HD High Definition. 16

HDMI High-Definition Multimedia Interface. 2

HDTV High Definition Television. 32

HEVC High-efficiency Video Coding. 30

HW Hardware. 58

I/O Input/Output. 42

IP Internet Protocol. 2

IT Information Technology. 2

KLT Karhunen–Loeve Transfor. 20

LAN Local Area Network. 34

LCD Liquid Crystal Display. 13

LTE Long-Term Evolution. 4

MDDT Mode-Dependent Directional Transform. 24

MPEG Moving Picture Experts Group. 30

NAL Network Abstraction Layer. 46

NDI Network Device Interface. 2, 34

OpenCL Open Computing Language. 61

OS Operating System. 42

PAL Phase Alternate Line. 10

PC Personal Computer. 53

PCIe Peripheral Component Interconnect express. 2

PTP Precision Time Protocol. 33

QCIF Quarter Common Intermediate Format. 52

RAII Resource Acquisition Is Initialization. 76

RGB Red Green Blue. 12

RGBA Red Green Blue Alpha. 12

RTP Real-time Transport Protocol. 46

SD Signal Degrade. 53

SDI Serial Digital Interface. 33

SDK Software Development Kit. 38

SHQ SpeedHQ. 35

SM Streaming Multiprocessors. 63

sRGB standard Red Green Blue. 15

SSD Solid-State Drive. 76

SVC Scalable Video Coding. 11

TV Television. 2

UHD Ultra High Definition. 31

USB Universal Serial Bus. 34

V4L `Video4Linux`. 43

V4L2 `Video4Linux2`. 43

VCEG Video Coding Experts Group. 31

VSCode Visual Studio Code. 39

Glossary

Artificial Intelligence is the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings. It is also referred to as **AI**. 3

ASIC stands for **A**pplication-**S**pecific **I**ntegrated **C**ircuit. It is an integrated circuit chip designed for a specific purpose, such as chip designed to run in a digital voice recorder. 42

bandwidth is a measure of how much data can be transferred per unit of time, generally measured in bits per second). To simplify, it measures size. 5

BMP whose full name is Microsoft Windows **B**it**m**a**p** Format, is one of the simplest format that stores pixels as an array of points and manages colors either in true color or through an indexed palette. 66

bootloader is a computer program that is responsible for booting a computer. In other words, it allows starting a computer. It is also called bootstrap loader [25]. 38

buffer is a data area shared by hardware devices or program processes. The buffer allows each device or process to operate without being held up by the other. 28

compilation is the process the computer takes to convert a high-level programming language into a machine language that the computer can understand. 41

context switch is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point. This allows multiple processes to share a single OS-level thread, and is an essential feature of a multitasking operating system. 45

deep learning is a machine learning sub-field concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. 5

demux is the process of reading a multi-part stream and saving each part (audio, video, and subtitles (if any)) as a separate stream. 56

DMA stands for **D**irect **A**ccess **M**emory. It is a feature of computer systems that allows certain hardware subsystems to access main system memory independently of the CPU. 43

driver is a set of files that tells a piece of hardware how to function by communicating with a computer's operating system. It is also called device driver. All pieces of hardware require a driver to abstract hardware details to the operating system, from the internal computer components, such as a graphics card, to the external peripherals, like a printer [51]. 38

entropy coding is a lossless data compression scheme that is independent of the specific features of the medium. 22

errno is a global variable accessible from any C++ application and that contains the code of the last error that was triggered in the Linux kernel. 45

file descriptor (FD) is a unique identifier for a file in UNIX conventions. 44

FPGA stands for **F**ield-**P**rogrammable **G**ate **A**rray. It is an integrated circuit designed to be (re)programmed. 2

frame is a single image in a sequence of pictures. 11

GPIO port stands for **G**eneral-**P**urpose **I**nput/**O**utput. It handles both incoming and outgoing digital signals. As an input port, it can be used to communicate to the CPU the ON/OFF signals received from switches, or the digital readings received from sensors. 6

GPU stands for **G**raphics **P**rocessing **U**nit. It is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device [52]. 3

I/O control is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls. 43

IDE stands for **I**ntegrated **D**evelopment **E**nvironment. It is a software application that provides comprehensive facilities (source code editor, build automation tools, a debugger, and others) to computer programmers for software development. 38

Internet of Things describes physical objects (or groups of such objects) with sensors, processing ability, software, and other technologies that connect and exchange data with other devices and systems over the Internet or other communications networks. It is also noted as **IoT**. In the consumer market, **IoT** technology is most synonymous with products pertaining to the concept of the *smart home*, including devices and appliances (such as lighting fixtures, thermostats, home security systems, cameras, and other home appliances) that support one or more common ecosystems, and can be controlled via devices associated with that ecosystem, such as smartphones and smart speakers [56]. 3

kernel
- is, in the context of an operating system, the core component of an OS that manages operations of computer and hardware. It basically manages operations of memory and CPU time. Kernel acts as a bridge between applications and data processing performed at hardware level using inter-process communication and system calls [24].
- is, in the context of `CUDA`, is a function that gets executed on GPU

. 38, 40, 89

latency is a measure of the delay in transferring some data between two nodes, generally measured in milliseconds. To simplify, it measures speed. i, 4

Linux is an operating system, such as Windows or Mac OS X. 3

machine learning is a branch of **A**rtificial **I**ntelligence (**AI**) and computer science which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy [59]. 42, 87

machine vision is a branch of **A**rtificial **I**ntelligence (**AI**) used, usually in industry, to provide imaging-based automatic inspection and analysis for applications such as automatic inspection, process control, and robot guidance [60]. 2

mmap s a UNIX system call to map files or devices into memory when dealing with Video4Linux, for operations such as I/O. . 76

mux is the process of combining inputs like video and audio. It then compresses these inputs into a container file. 56

NVIDIA NVIDIA Corporation is an American multinational technology company. It is a software and fabless company which designs **G**raphics **P**rocessing **U**nits (**GPUs**), **A**pplication **P**rogramming **I**nterface (**APIs**) for data science and high-performance computing as well as **S**ystem **o**n a **C**hip units (**SoCs**) for the mobile computing and automotive market. NVIDIA is a global leader in artificial intelligence hardware and software from edge to cloud computing [62]. 1

OpenCV is an open-source computer vision and machine learning software C++ library which implements more than 2000 optimized algorithms. . 74

overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task [63]. 29

PCB stands for **P**rinted **C**ircuit **B**oard. It is a board with circuits that connect electronic components together. 2

pixel is the smallest controllable element of a picture represented on the screen. It is characterized by a finite, discrete quantities of numeric representation for its intensity or gray level that is an output from its two-dimensional functions fed as input by its spatial coordinates denoted with x, y on the x-axis and y-axis, respectively. 12

SECAM stands for **S**ystem **E**ssentially **C**ontrary to **A**merican **M**ethod is a French video format standard that uses a refresh rate of 50Hz. 16

sparse matrix is a matrix containing almost exclusively zeros. 21

SSH stands for **S**ecure **SH**ell. It allows connection to a remote machine (acting as a server) from another machine (acting as a client) via a secure link to transfer files or commands securely. 39

system call is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed. A system call is called syscall [24]. 76, 88

transcode is the process of converting from one form of coded representation to another, for example, transcode an AVC stream into an HEVC stream. 56

Ubuntu is an operating system, such as Windows or Mac OS X, based on Linux kernel. 38

UNIX was earlier known to be UNICS, which stands for **UN**iplexed **I**nformation **C**omputing **S**ystem. UNIX is a popular operating system, first got released in 1969. UNIX is a multi-tasking, powerful, multi-user, a virtual OS which could be implemented on a variety of platforms. It is the building block of the variety of OS based on the Linux kernel [69]. 76, 87

video overlays is an image or animation that can be added to videos which will appear, for example, on the bottom of the video screen over the video. 3

Virtual Reality is a technology that superimposes a computer-generated image on a user's view of the real world, thus providing a composite view. It is also referred to as **AR**. Pokemon GO is one of the many AR applications. 3

Virtual Reality is a simulated experience that can be similar to or completely different from the real world. Applications of virtual reality include entertainment (particularly video games), education (such as medical or military training) and business (such as virtual meetings) [71]. It is also noted as **VR**. 3

List of Figures

Appendix A

Thesis statement

Annexe A : Description du stage

En cas de contradiction entre la présente Annexe A et la Convention de stage, cette dernière prévaudra.

Titre : « Utilisation d'une plateforme NVidia Jetson Xavier comme système de diffusion multimédia »

Cible : Sciences Informatiques / Ingénieur Civil en Informatique

Référents : Julien Jemine

Centre de compétence interne: Streaming & Embedded Software

Description du travail :

L'objectif du travail est de réceptionner un ou plusieurs flux multimédia sur IP (de type NDI ou NDIHX) sur une plateforme NVidia Jetson Xavier, de le(s) décoder, et de le(s) transmettre sur une ou plusieurs sorties HDMI.
L'étudiant devra prendre en main la plateforme et son environnement de développement associé.
La vidéo est encodée en MPEG-2 ou en H.264. L'étudiant devra utiliser les accélérations hardwares de la plateforme pour la décoder efficacement.
La sortie HDMI se fera au travers d'une carte PCIe Deltacast que l'étudiant devra manipuler au travers de son API VideoMaster.
S'il dispose de suffisamment de temps, l'étudiant étudiera également le flux inverse : réception de flux en HDMI, encodage, et émission d'un flux réseau sur IP.

9

De plus l'étudiant devra faire l'implémentation du décodage d'un flux NDI standard en CUDA.

Il aura également la possibilité d'ajouter des overlays sur le flux de sortie. Ces overlays pourraient provenir soit d'une autre entrée vidéo, soit d'un buffer GPU, soit d'une technologie d'interface graphique tournant sur le Linux embarqué.

Appendix B

Frame quality for different video coding format

(a) H.264

(b) H.265

(c) MPEG-2

(d) MPEG-4

(e) VP8

(f) VP9