NTNU

Norwegian University of
Science and Technology

Real-Time JPEG2000 Video Decoding on General-Purpose Computer Hardware

Erlend Halsteinli

Master of Science in Electronics
Submission date: June 2009
Supervisor: Andrew Perkis, IET
Co-supervisor: Odd Inge Hillestad, T-VIPS AS

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

JPEG2000[1] is a powerful image compression standard and amongst other used within Digital Cinema and high resolution video transport. The coding efficiency of JPEG2000 comes at the expense of significantly increased algorithmic complexity compared to older standards, e.g. JPEG. This has limited the availability of cost-effective, high-performance solutions targeting general-purpose computer hardware.

The work of this Master Thesis is to study JPEG2000 and develop a real-time decoder for high resolution JPEG2000 video. The codestream shall originate from a T-VIPS TVG430 video gateway and be received through an IP interface. The decompressed video frames shall be presented to the user immediately after decoding in a graphical user interface. Real-time demands shall be fulfilled by including the GPU in the decoder pipeline[2].

The implementation shall be done using C++ and CUDA.

[1] - D Taubman and M Marcellin, JPEG2000: Image compression fundamentals, standards and practice, Kluwer Academic Publishers, 2001.
[2] - G. Shen et al, Accelerate Video Decoding With Generic GPU, IEEE Transactions on Circuits and Systems for Video Technology, 2005.

Assignment given: 15. January 2009
Supervisor: Andrew Perkis, IET

Abstract

There is widespread use of compression in multimedia content delivery, e.g. within video on demand services and transport links between live events and production sites. The content must undergo compression prior to transmission in order to deliver high quality video and audio over most networks, this is especially true for high definition video content.

JPEG2000 is a recent image compression standard and a suitable compression algorithm for high definition, high rate video. With its highly flexible embedded lossless and lossy compression scheme, JPEG2000 has a number of advantages over existing video codecs. The only evident drawbacks with respect to real-time applications, are that the computational complexity is quite high and that JPEG2000, being an image compression codec as opposed to video codec, typically has higher bandwidth requirements.

Special-purpose hardware can deliver high performance, but is expensive and not easily updated. A JPEG2000 decoder application running on general-purpose computer hardware can complement solutions depending on special-purpose hardware and will experience performance scaling together with the available processing power. In addition, production costs will be none-existing, once developed.

The application implemented in this project is a streaming media player. It receives a compressed video stream through an IP interface, decodes it frame by frame and presents the decoded frames in a window. The decoder is designed to better take advantage of the processing power available in today's desktop computers. Specifically, decoding is performed on both CPU and GPU in order to decode minimum 50 frames per second of a 720p JPEG2000 video stream. The CPU executed part of the decoder application is written in C++, based on the Kakadu SDK and involve all decoding steps up to and including reverse wavelet transform. The GPU executed part of the decoder is enabled by the CUDA programming language, and include luma upsampling and irreversible color transform.

Results indicate that general purpose computer hardware today easily can decode JPEG2000 video at bit rates up to 45 Mbit/s. However, when the video stream is received at 50 fps through the IP interface, packet loss at the socket level limits the attained frame rate to about 45 fps at rates of 40 Mbit/s or lower. If this packet loss could be eliminated, real-time decoding would be obtained up to 40 Mbit/s. At rates above 40 Mbit/s, the attained frame rate is limited by the decoder performance and not the packet loss. Higher codestream rates should be endurable if reverse wavelet transform could be mapped from the CPU to the GPU, since the current pipeline is highly unbalanced.

Preface

This Master's thesis was written during the spring semester of 2009 at the Norwegian University of Science and Technology (NTNU). The thesis was carried out at the Department of Electronics and Telecommunication (IET) in collaboration with the company T-VIPS. The purpose of the thesis was to develop a JPEG2000 video decoder, obtaining minimum 50 frames per second when decoding a 720p video codestream. The application should utilize the GPU in order to obtain real-time decoding.

I would like to thank my supervisor, professor Andrew Perkis, for his support throughout the work on my thesis. I would also like to thank T-VIPS, especially my co-supervisor Odd Inge Hillestad for his constructive feedback during the writing process and Darren Starr for suggesting the GPGPU approach to JPEG2000 decoding.

Trondheim, June 2009
Erlend Halsteinli

Contents

List of Figures

List of Abbreviations

5/3	CDF wavelet with integer taps. Mostly used in lossless coding.
720p	Video resolution: 1280x720 pixels progressive.
9/7	CDF wavelet with floating point taps. Used in lossy coding.
CDF	Cohen-Daubechies-Feauveau, family of biorthogonal wavelets.
Codestream	Single JPEG2000 encoded frame.
CPU	Central Processing Unit.
CPU Memory	CPU DRAM, located on motherboard. Not cache.
CUDA	Programming language, used to perform GPGPU.
DCT	Discrete Cosines Transform.
Device	CUDA expression, referring to the GPU.
Device Memory	See *GPU Memory*.
DWT	Discrete Wavelet Transform.
FBS	Filter Bank Scheme, a way to perform wavelet transform.
FIFO	First-In, First-Out.
FLOP	FLoating point OPerations.
fps	frames per second.
GPGPU	General-Purpose computing on Graphics Processing Units.
GPU	Graphics Processing Unit.
GPU Memory	GPU DRAM, located on graphics card.
GUI	Graphical User Interface.

HD	High Definition.
Host	CUDA expression, referring to the CPU.
Host Memory	Se *CPU Memory*.
ICT	Irreversible Color Transform.
IDWT	Inverse Discrete Wavelet Transform.
IP	Internet Protocol.
JPEG	Joint Photographic Experts Group. Both a committee with members from ISO/IEC and ITU-T, and an image compression standard from the same committee.
JPEG2000	Image compression standard from the JPEG committee.
Kakadu	Popular JPEG2000 SDK.
Kernel	CUDA function, concurrently executed on multiple data.
LS	Lifting Scheme, a way to perform wavelet transform.
Marker segment	Building block in the JPEG2000 codestream.
MSB	Most Significant Bit.
Mutex	Special variable in programming language with atomic get and set methods.
MXF	Material eXchange Format.
PBO	Pixel Buffer Object.
Pipeline	A set of data processing modules connected in series, where the output of one module is the input of the next one.
RGB	Color space, mostly used in computer systems.
RTP	Real-time Transport Protocol.
SD	Standard Definition.
SDK	Software Development Kit.
SIMT	Single Instruction, Multiple Threads.
TVG430	Video Gateway from T-VIPS, employing JPEG2000 coding and IP+RTP+MXF encapsulation.
UDP	User Datagram Protocol.
YCbCr	Color space, mostly used in digital video coding.

Chapter 1

Introduction

Communication between humans has historically been based on speech, writing and signs. Our everyday communication habits have evolved over the last decades to include other types of communication, like digital images, music and videos. The introduction of these new communication types has not been at the expense of existing ones, and people communicate more today than ever before. Examples include sending of emails with attachments, instant messaging, video conferencing, online gaming and online social- and professional networking. This variety of communication mediums are the results of the technological progress also made over the last decades.

Not all communication types used today are based on new concepts. Rather, they have been renewed by the employment of new technology and thereby increased their market appeal. Video streaming is an example of the latter, since it has been used within digital television distribution since the first broadcasts. In streaming applications the content is constantly received by, and often presented to, an end-user. Streaming does in this context refer to the content delivery method, rather than to the content type or channel type itself, but the content is mostly distributed over television broadcast networks and telecommunication networks.

Both professionals and consumers benefit from high quality video streaming services. Examples of such services are the movement of uncut content from a live event to a production site, or the transmission of a TV show from a content provider to a number of consumers. All video in real-time streaming applications must in practice undergo some sort of compression before transmission, due to limited and expensive transmission bandwidth. Modern compression algorithms provide high compression ratios and rich functionality while retaining high visual quality, but have higher computational complexity compared to previous generations of algorithms. E.g. JPEG2000 and H.264 are algorithmically more complex than their predecessors JPEG and H.263[1, 2]. This computational complexity can compose a problem in real-time decoding of streamed high definition video. Special-purpose hard-

ware will deliver the needed computing power, but not all scenarios embrace such a solution, since special-purpose hardware is expensive and can quickly become outdated. A software decoder targeting general-purpose computer hardware will supplement solutions depending on special-purpose hardware, automatically experience performance scaling with the available processing power and has zero production costs once developed, but only if the real-time requirements can be fulfilled.

A considerable amount of work and effort has been put into optimization of multimedia processing with respect to throughput on general-purpose computer hardware[3, 4, 5]. Various techniques are described, but programmers have usually been restricted to in-line assembly, intrinsic functions or specialized libraries when optimizing CPU executed code[6]. Recently, the field called General-Purpose Computing on GPUs (GPGPU)[7] has received increased attention. GPGPU provides a different approach to accelerating multimedia processing, since tasks can be handed over from the CPU to the GPU. Operations such as matrix multiplication[8], Discrete Cosines Transform[9], Discrete Wavelet Transform[10] and Fast Fourier Transform[11] have already been successfully mapped to the GPU. In [12], Wong *et al.* enhanced the performance of the JPEG2000 codec JasPer with GPU based wavelet transform. The first attempt to use the graphics pipeline to accelerate video decoding, was reported by Shen *et al* in [13]. In all, it is most likely that GPGPU can improve performance in video decoding applications.

The aim of this work is to obtain real-time decoding of a JPEG2000 video stream, targeting occasions where special-purpose decoding hardware is unsuitable. The application shall receive the compressed video stream through an IP interface, and the decoding shall be done on general-purpose computer hardware. The application shall be able to decode minimum 50 frames per second at a resolution of 1280x720 pixels. Real-time requirements shall be met by pipelining the CPU and the GPU, and handing parts of the decoding process over from the CPU to the GPU.

The thesis is organized as follows. Chapter 2 provides background information on video transport, a specification of the software decoder and its user scenarios, in addition to a short overview of existing decoder applications. Chapter 3 presents relevant theory on image compression and video compression, while chapter 4 contains information regarding General-Purpose Computing on GPUs. Chapter 5 presents details regarding the development process and chosen software architecture. The obtained decoder performance is presented in chapter 6 and discussed in chapter 7, while the work is concluded in chapter 8. Future work is outlined in chapter 9. The source code is given in appendix E.

Chapter 2

Background

This chapter provides background information on the transportation of HD video and a specification of the software decoder containing the necessary high-level engineering requirements. The chapter rounds off with a short overview of existing video decoders with JPEG2000 capabilities.

2.1 Transmission of HD Video

Streaming is a popular delivery method for video, as mentioned introductorily. This section inspects the video transmission chain from a wider perspective.

The market for HD content has grown together with the number of sold High Definition (HD) TV sets, but improved broadcast infrastructure may be needed for consumers to receive HD content. Figure 2.1 shows a simplified diagram of the transmission chain, with the content production site as source and the consumer as sink. A complete specification of the transmission protocols between entities in this chain, is far beyond the scope of this work. It has therefore been abstracted into two parameters, namely compression algorithm and network protocol.

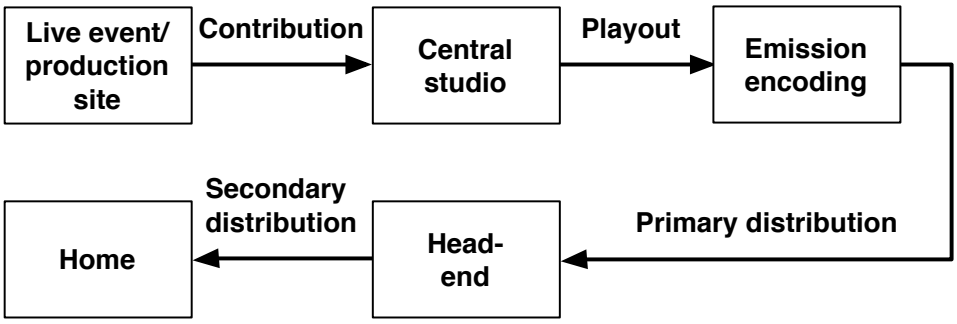


Figure 2.1: The content transmission chain.

Both contribution and playout address the medium-independent point-to-point movement of broadcast material between two entities, while distribution usually is a one-to-many transmission. In this model the content is produced at some live event or production site, and transmitted (contribution) to a central studio where it is integrated into a finished show. This program material is then sent (playout) to an emission encoder where it is applied an appropriate encoding scheme and distributed to cable/satellite/DTT(Digital Terrestrial Television) head-ends where a number of channels are merged into the secondary distribution stream transferred to the end-consumer. If the content not originates from a live event, it may be stored at several points in the chain. Standard Definition (SD) contribution has the last years usually been done by compressing the stream with MPEG-2 to about 10-15 Mbit/s and transmitting it over a dedicated line, e.g. ATM. As HDTV demands about four times the bit rate of a SD signal, the broadcasters have two solutions: either scale the existing solutions to handle the increased bit rate or employ new technology. We will consider the latter case, where compression algorithm and encapsulation method will be investigate.

Multiple compression algorithms for image content exists, e.g. MPEG 2, MPEG 4, VC-1 and JPEG2000. The preferred algorithm must yield a high compression ratio while retaining high fidelity, but HD video compression in professional applications impose strict requirements. Transmission errors should be short-lived and not produce blocking artifacts. The absence of inter-frame coding would ease the postproduction process and lower the encoder/decoder latency. JPEG2000[14] is very suitable as HD video compression algorithm, since it is developed to produce a compact, high quality representation of single images and covers the above mentioned demands. It should be noted that Digital Cinema Initiative (DCI) has chosen JPEG2000 as compression algorithm for motion pictures in their Digital Cinema System Specification[15], and that their requirements were similar at many points.

The employed network protocol must be reliable and cost-effective, but also flexible and scalable. It is not unlikely that Internet Protocol (IP) will become more and more popular for video transportation. First, IP can easily encapsulate the compressed codestream. Second, it is currently a convergence towards IP on many fields, e.g. data, voice, surveillance and video conferencing. Some of the reasons behind this may be the low infrastructure cost and high bandwidth, implying lower cost per bandwidth unit. The need to operate and maintain different networks vanishes, since most companies already have IP based data networks.

In the light of the above discussion, it seems clear that JPEG2000 over IP offers a number of advantages in HD video transportation. High quality content can easily be moved between production sites and delivered to consumers. The scalability within IP and JPEG2000 makes it a competitive solution, now and in the future.

2.2 Software Decoder Specification

This section specifies the engineering requirements for a software implementation of a JPEG2000 video decoder running on general-purpose computer hardware. The purpose of this software decoder is real-time playback of a JPEG2000 video stream from network to display, targeting applications where a special-purpose hardware decoder is unsuitable. The application shall be able to decode minimum 50 frames per second (fps) at a resolution of 1280x720 pixels. Thus, in the following, real-time decoding will refer to 50 fps or more. The video stream shall originate from a T-VIPS TVG430[16] video gateway, but the decoder core will be generalizable to any JPEG2000 code stream. The application shall perform the following tasks concurrently:

- Receive UDP/RTP packets.
- Reconstruct JPEG2000 codestreams from multiple UDP/RTP payloads.
- Decode JPEG2000 codestreams.
- Display decoded video frames.

As described in section 2.1, JPEG2000 applies well to high quality video. Because of the computational load associated with JPEG2000 encoding and decoding, it can be challenging to fulfill real-time requirements. Modern GPUs have massive processing power deployable in processing with a high degree of parallelization. By pipelining the CPU and GPU within the JPEG2000 decoder and handing decoding tasks over from the CPU to the GPU, sufficient throughput should be obtained and real-time requirements met. Three parts of the decoding process stand out as primary candidates for processing on the GPU, i.e. reverse wavelet transform, chroma upsampling and color space transform. The deployment of the GPU within the decoder is not only motivated by the GPUs ability to perform certain tasks faster than the CPU, but also the possibility to pipeline[1] the CPU and GPU.

2.2.1 User Scenarios

Three user scenarios for the software decoder have been identified and named Monitoring, Debugging and Distribution. The first two can be seen in figure 2.2 and the last in figure 2.3. The location of the software decoder can be seen in both figures as red boxes.

[1] A pipeline is a set of data processing modules connected in series, where the output of one module is the input of the next one. The modules in a pipeline are often executed in parallel and buffers are often inserted between the modules.

Monitoring

The primary objective in the *Monitoring* scenario is transmission of video content from source to drain, which is equal to the contribution and playout processes described in section 2.1. This can e.g. be handled by two TVG430 as seen in the dashed box in figure 2.2. In such a scenario the broadcaster or transmission entity would want to have continuous supervision and monitoring of the transmission. I.e. visually control the presence and quality of the transferred content. The person supervising the transmission does not need to be at neither the source or the drain, rather at an arbitrary location connected to the IP network, as seen in figure 2.2. A hardware device may be more appropriate for high quality monitoring, but a software solution will be mobile and more cost- and space efficient.

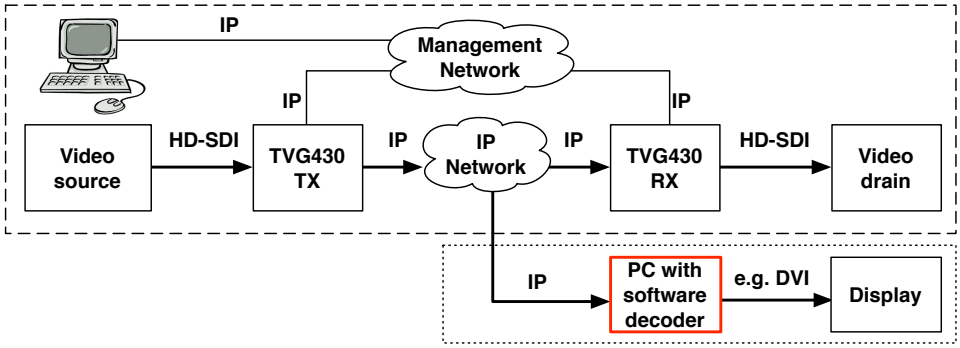


Figure 2.2: A usual setup with two TVG430 is shown within the dashed box, while the dotted box contains the software decoder placing. Both boxes compose the Monitoring and Debugging scenarios.

Debugging

When a video link breaks down, it is desirable to quickly locate the source of error, which forms the basis for the *Debugging* scenario. A technician with access to the IP network and a laptop with the decoder software, can quickly limit the number of possible error sources. If the technician is able to receive and decode the video stream, the decoder is most likely the source of error. But if the technician is unable to receive the video stream, most likely the error source is the encoder or network infrastructure. Setup for the Debugging scenario is the same as for Monitoring.

Distribution

In the *Distribution* scenario HD material is delivered to multiple consumers. An increasing number of consumers will have high-speed fiber connections in

densely populated areas, even at the last mile[2]. Thus, high quality content may be received through this connection. Each user must be connected to the IP network and run the software decoder as shown in figure 2.3. The consumer may present the decoded video stream at any screen of choice, e.g. computer display or television. Software upgrades are easy to distribute and no additional hardware is need if the consumer already has a suitable computer.

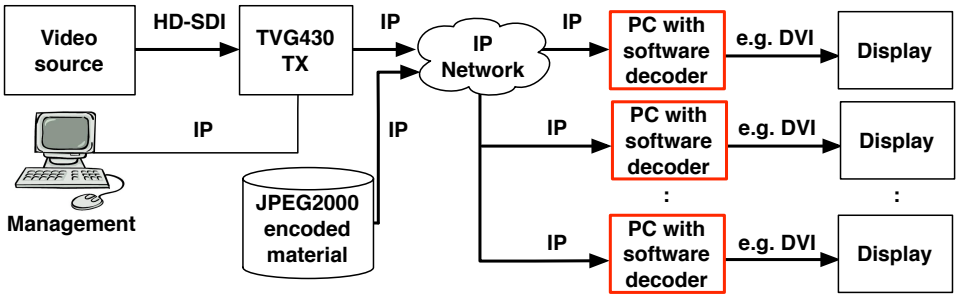


Figure 2.3: The Distribution user scenario.

2.3 Existing Software Decoders

Known applications capable of decoding JPEG2000 video are listed in table 2.1. Both Aware and Kakadu provide their products as a proof of concept for their JPEG2000 SDKs, and not as commercial applications. All applications listed in table 2.1 are based on CPU decoding from local storage. None of the applications can handle streamed JPEG2000 video or utilize the GPU in the decoding process.

Table 2.1: Applications Supporting JPEG2000 Part 1

Company	Program
Aware	Sample application
Kakadu	kdu_vex_fast

[2]The final leg from a communications provider to a customer, typically the distance from the nearest switching central to your house.

Chapter 3

Image and Video Coding

This chapter considers digital image and video coding, which include techniques to accurately and compactly represent the given data. Many considerations and trade-offs have to be made when designing a coding scheme, but the applied compression technique is most likely the biggest design issue. Image and video compression are a special case of data compression, and reduce the usage of limited resources such as hard drive capacity or transmission bandwidth. Depending on the area of application, the compression process may be lossless or lossy. In the lossless case, internal signal dependencies are removed in order to obtain a representation length close to the entropy of the original signal, but the original can still be perfectly reconstructed. The Lempel-Ziv[17] algorithm and its many variations are amongst the most popular algorithms for lossless storage. If the perfect reconstruction requirement is eased, higher compression ratios can be attained, but noise is inevitably introduced. JPEG and the various MPEG compression standards are examples of popular lossy coding schemes. The amount of acceptable noise will wary with the area of application, but compression beyond the lossless case is possible even if visual degradation is unacceptable. A representation length that is shorter than the entropy of the original may by achieved, by exploiting the fact that the human observation system only can absorb a limited amount of information. E.g. the eye is much more sensitive to variations in luminance, than to variations in color, and therefor less resources may be spent on color coding without any visual difference. Image and video compression is usually lossy, where the goal typically is to obtain minimal bit rate at a given distortion or to obtain minimal distortion at a given bit rate. High performance compression is obtained by utilizing the two basic properties explained above, namely signal redundancy and visual redundancy. Besides the introduced noise in lossy compression, the only evident drawback connected to data compression is the fact that some schemes are quite complex and time consuming, and can therefor be problematic in applications with limited processing capabilities or real-time requirements.

3.1 JPEG2000

JPEG2000 is a still image compression standard developed by the JPEG committee. Supplementing the widely used JPEG standard, it is a highly flexible embedded lossless and lossy coding scheme and supports a wide set of features. *JPEG2000 Core coding system*, was given status as an international standard in december 2000 and is presented in this section. There are several books and articles giving an overview of the standard, this presentation is mainly based on [18, 19, 20].

The compression standard can be decomposed into three main parts as shown in figure 3.1. First is a *preprocessing* step, then *core processing* followed by *bitstream formation*. The figure also shows the operations within the core processing. In accordance with JPEG and MPEG tradition, only the codestream syntax and decoder are standardized. This enables encoder improvements after standardization and competition between different designs with regards to cost, performance and complexity. JPEG2000 encoding is considered in the following sections, because it gives a better understanding about how compression is obtained. Decoding involves the same steps in reversed order.

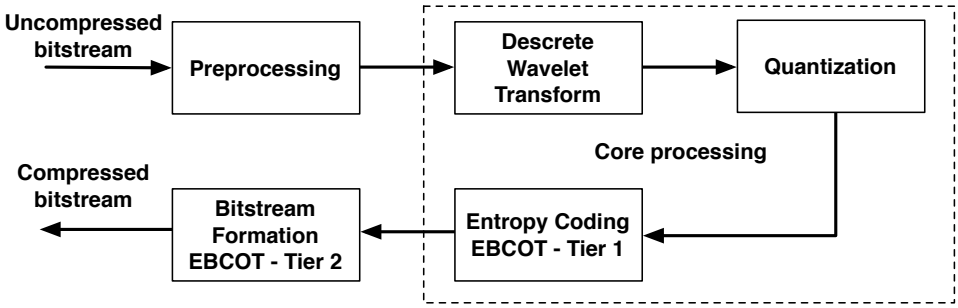


Figure 3.1: JPEG2000 block diagram.

3.1.1 Preprocessing

Tiling

The preprocessing starts with an image *tiling*, were the image is partitioned into rectangular non-overlapping sections covering the entire image. All tiles have the same size, with exceptions only at the image boundaries since the image resolution may not be an integer multiple of the tile size. The tile size is upper limited by the image dimensions, i.e. one tile covering the entire image. Furthermore, tiling reduces the memory requirements since each tile is processed individually and only the tile currently being processed must reside in memory. Tiling will unfortunately introduce artifacts at low rates compared to processing the entire image as one tile. In the subsequent

sections we assume that the image consist of a single tile, since the extension to multiple tiles is trivial.

DC Level Shift

Following after the tiling is a *DC level shift*. All unsigned samples are shifted down so they are symmetrically distributed around zero. It should be noted that this has no effect on the coding efficiency, but merely is done to simplify certain implementation subjects, such as numerical overflow and arithmetic coding.

Color Transform

The last preprocessing step is an optional *color transform* of the first three image components. Depending on the following wavelet transform one of two color transforms may be applied. Both assumes the three first components belongs to the Red-Green-Blue (RGB) color space. The first is the Irreversible Color Transform (ICT), which is a RGB to YCbCr transform used with lossy coding. The second is the Reversible Color Transform (RCT), which is an approximate RGB to YUV transform and may be used with both lossy and lossless coding. Both transforms are linear, and in order to apply them the components must have the same bit depth and subsampling. Both color transforms achieve color decorrelation for efficient compression and an appropriate color space for quantization, i.e. quantization in the transformed color space gives less visual artifacts than quantization in the original color space.

3.1.2 Core Processing

Each image component is processed individually within the following core processing steps.

Discrete Wavelet Transform

The *Discrete Wavelet Transform* (DWT) may be considered as both a subband technique and a transform coding method, producing a multiresolution representation of the original sequence. A one dimensional wavelet transform gives a decomposition of the given input signal into two resolution bands, called *detail* and *approximation*. The approximation signal is a coarse-grained representation of the input, while the detail signal contains the high-frequency information not present in approximation. Together they can reconstruct the original signal perfect, disregarding rounded floating-point calculations. Traditionally, DWT has been computed with the *Filter Bank Scheme* (FBS), which include convoluting the input signal with the

impulse response of either a low-pass or a high-pass filter, followed by critical down-sampling. This scheme can be seen in figure 3.2.

The *Lifting Scheme* (LS) is an approach proposed more recently[21]. Its main advantage over FBS is that it exploits the redundancy between the high-pass and low-pass filter, and thereby reduces the number of arithmetic operations. For long filters, FBS asymptotically tends to require twice as many operations as LS[22]. Computing DWT with the lifting scheme, include a number of *prediction* and *update* steps. A prediction step (p) consists of predicting each odd sample as a linear combination of the even samples and subtracting the prediction from the odd sample to form the prediction error. An update step (u) consists of updating the even samples by adding to them a linear combination of the already modified odd samples. The number of needed lifting steps per DWT level will depend on the wavelet in question. A lifting procedure with two prediction and two update steps can be seen in figure 3.3.

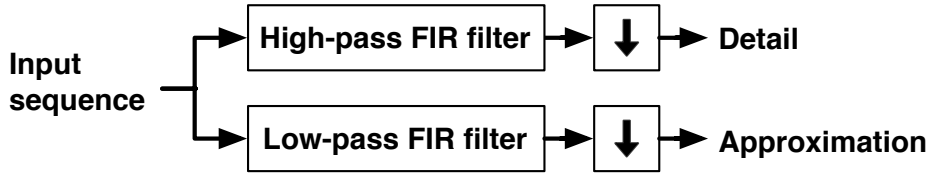


Figure 3.2: One dimensional Wavelet Transform computed with filter bank.

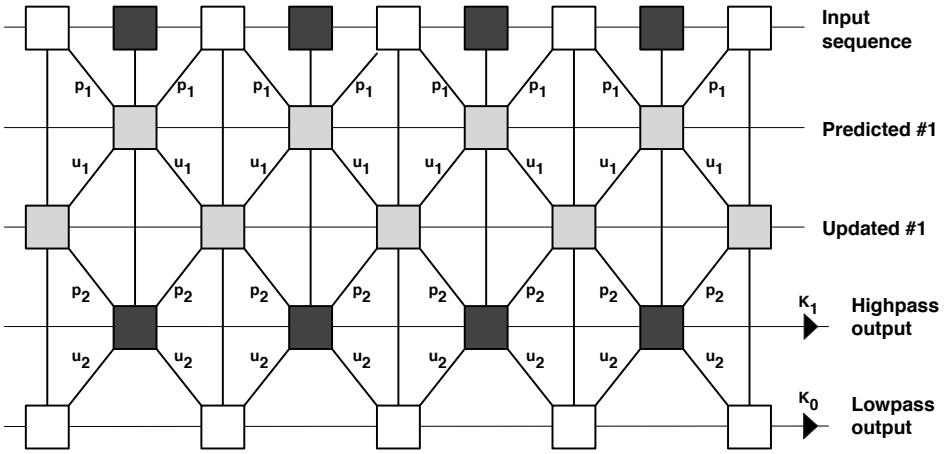


Figure 3.3: One dimensional Wavelet Transform computed with lifting. Two prediction (p) and update (u) steps are performed.

FBS and LS are mathematically equivalent with respect to the output signals, although the intermediate signals are different. The reduced number of arithmetic operations in the lifting scheme makes it superior on many platforms. E.g, a performance comparison between FBS and LS on a DSP,

concluded that LS always performs better with respect to execution time[23]. A similar result was presented in[24]. The reduction in arithmetic operations does however introduce data dependencies in the signal being transformed, which can become a bottleneck on highly parallel platforms. Wavelet transformation performed on modern GPUs would benefit from avoiding data dependencies, and should therefor use the filter bank scheme[25].

In JPEG2000 part 1, two wavelets are chosen from the diversity of existing wavelets, namely the Cohen-Daubechies-Feauveau (CDF) 9/7 for lossy coding and the CDF 5/3 for lossless coding. A one level 5/3 wavelet transformation can be performed with one p and one u step, while a one level 9/7 wavelet transformation can be performed with two p and two u steps. The lifting coefficients for the 9/7 wavelet are given in table 3.1, while the lifting steps are illustrated in figure 3.3.

Table 3.1: Lifting coefficients for the CDF 9/7 wavelet.

p_1	-1.586134342059924
p_2	-0.052980118572961
u_1	0.882911075530934
u_2	0.443506852043971
$K_1 = 1/K_0$	1.230174104914001

The wavelet transform is applied both in horizontal and vertical direction in JPEG2000. The transformation order is indifferent, since the transformation is a linear operation, i.e., horizontal processing may precede vertical processing or the other way around. Figure 3.4 and 3.5 show the step by step transformation of an image, where the first figure indicates the subband positions and the latter displays the actual transform output. After a one level, two dimensional wavelet transform, we obtain a four-split of the input as seen in figure 3.5b. The first letter in each subband name denote the horizontal filter and the second letter denote the vertical filter, for instance HL1 is horizontally high-pass filtered and vertically low-pass filtered. Note that most of the energy is concentrated at the low frequencies, i.e. in the LL subband, and that the other sub-bands only have energy at the detailed regions (high concentration of high frequencies). Since the LL1 subband contains most of the energy, it is applied a second transformation. The result can be seen in figure 3.5c. The LL subband can be transformed further, until the desired number of decomposition levels is obtained.

After transformation, the wavelet coefficients are specified to be stored in an interleaved scheme as seen in figure 3.6. Depending on the way the encoder produces wavelet coefficients, interleaving may be needed after the transformation is performed.

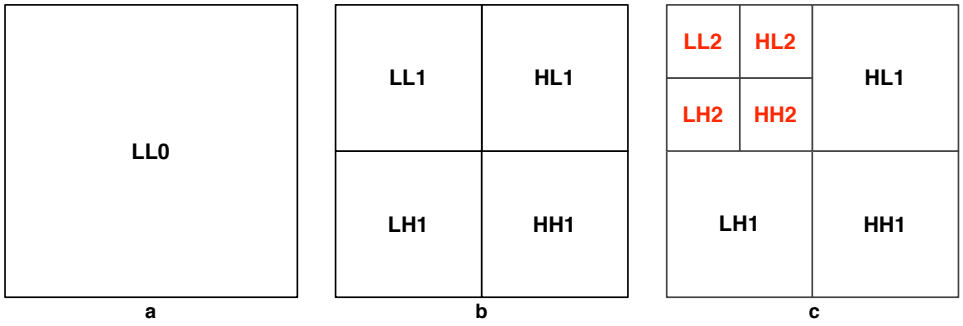


Figure 3.4: The subbands produced by a dyadic, two dimensional DWT. a) Image before transformation. b) Image after the first transformation step. c) Image after the second transformation step.

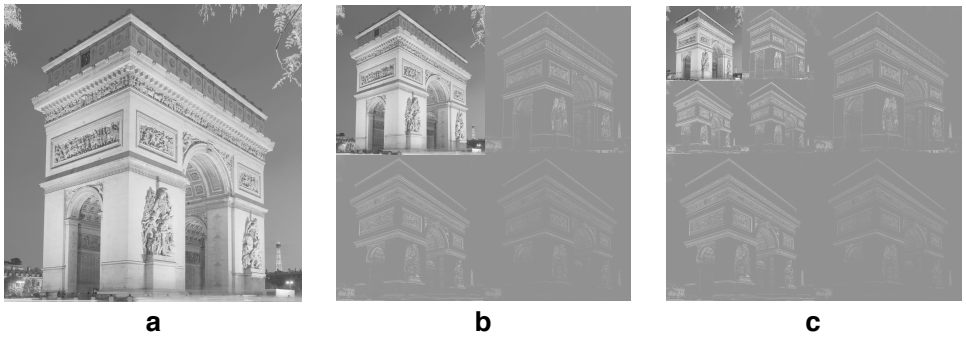


Figure 3.5: Two dimensional DWT using the 9/7 filter-bank. a) The image before transformation. b) The image after the first transformation step. c) The image after the second transformation step. Note that the only change from *b*, is that LL1 is replaced by LL2, HL2, LH2 and HH2, while HL1, LH1 and HH1 are unchanged.

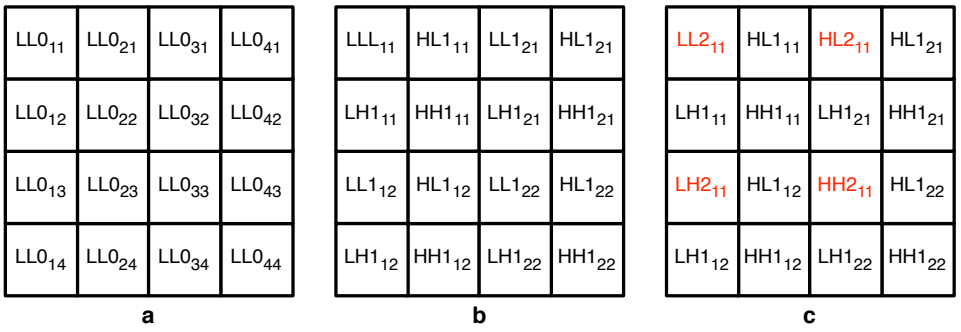


Figure 3.6: Section of the interleaved coefficients after two dimensional DWT. The three first letters in the coefficient name denote the subband, while the two numbers in subscript denote the coefficient index within the given subband. E.g, $LH1_{21}$ is a coefficient from column two and row one in the LH1 subband. a) The pixels values prior to transformation. b) The coefficients after first transform step. c) The coefficients after the second transformation step. Note that only the red coefficients have changed from *b*.

Quantization

After the DWT follows a straight-forward *quantization* procedure, namely an uniform scalar quantization with a central deadzone. This is shown in [26] to be rate-distortion optimal for a continuous signal with a Laplacian probability density function (pdf). Wavelet coefficients typically follows a Laplacian pdf, and can be approximated as continuos for high bit rates. The central deadzone is twice the step size, which may be different for each subband. An important property with this approach is that if subband b is quantized with M_b bits and step size Δ_b but only N_b bits is decoded(N_b<M_b), this is the same as quantizing with N_b bits and step size $\Delta_b * 2^{M_b - N_b}$. In other words we obtain SNR scalability without any additional coding cycles. In the case of lossless compression, the coefficients being quantized are integers and the quantizer step size is one.

EBCOT - Tier 1

Tier 1 is the first layer of the JPEG2000 coding engine called *Embedded Block Coder with Optimal Truncation* (EBCOT)[27], and performs source modeling and entropy coding of the quantized wavelet coefficients. EBCOT is second large difference in JPEG2000 compared to JPEG, and has at its core an adaptive arithmetic coder named the MQ-coder. This is a modification of the Q-coder[28] developed by IBM, that simplifies the probability estimation compared to usual block-coding because no joint-probability estimation is needed. The MQ-coder does not exploit any statistical dependencies between subband, most of all because this would reduce the bitstream flexibility and remove the ability to change progression order without transcoding.

In the same manner that the image was tiled in section 3.1.1, each subband is now compulsorily partitioned into *code-blocks*, containing e.g. 32x32 or 64x64 quantized wavelet coefficients. Each code-block is bit-plane coded starting with the most significant bit (MSB). Each bit-plane is coded in three passes, with the ability to truncate the bitstream after each pass. Each quantized wavelet coefficient has a binary state variable called *significance state*, which changes from zero to one(significant) when the first non-zero bit is found. The probability estimate for each bit is produced from its significance state and the significance state of its neighbors. The symbol and the MQ-coders state are used to encode the given symbol. Then the coders internal state is updated in order to refine probability estimates for the current context.

A deliberate redundancy in the MQ-coder results in that any two consecutive bytes of coded data are forced to lie in the range 0x0000 to 0xFF8F, leaving the range 0xFF90 to 0xFFFF free to represent unique marker[1] codes. This assists codestream parsing and improves error resilience.

[1] See section 3.1.4 for more on marker segments.

3.1.3 Bitstream Formation

EBCOT - Tier 2

The bitstream is formed in *Tier 2*, the second layer of EBCOT. For each code-block the output from Tier 1 is a single codeword and side information that indicates the valid truncation points and the given distortion at each truncation point. This information is used to find the optimal truncation points for a given target bit-rate, i.e. iteratively finding the (possibly truncated) codeword that gives the largest reduction in distortion relative to the codeword length. This post-encoding truncation scheme gives the advantage that no additional coding cycles are need to obtain the target bit rate or quality. Finding the optimal truncation points is a global optimization problem over all code-blocks and may be solved by Lagrange multipliers.

The smallest codestream building-block is the compressed code-block, which is a number of neighboring wavelet coefficients after quantization, entropy coding and truncation. A number of spatially consistent compressed code-blocks from each subband at a given resolution level form a *precinct* as seen in figure 3.7a. A number of coding passes for each code-block in the precinct forms the body of a *packet*, which is a quality increment at a spacial location for a given resolution level. In a similar fashion, one packet from each precinct at a given resolution level comprises a *layer*, which is a quality increment for the entire image at that resolution level. A number of layers make up the codestream, and the resulting structure can be seen in figure 3.7b.

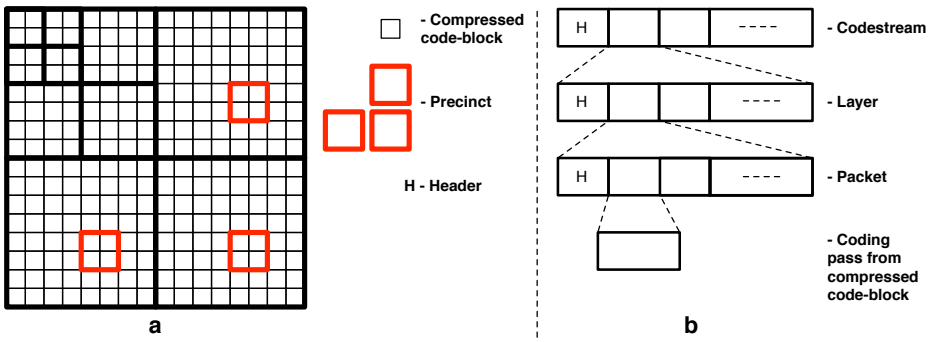


Figure 3.7: Codestream formation

By carefully choosing the order in which packets are included in the codestream, the desired *progression order* is obtained. The different progression orders are defined by the ordering of layer(L), component(C), position(P) and resolution(R). The allowed progression orders are RLCP, RPCL, PCRL, CPRL and LRCP, the last one expressed in pseudocode in listing 3.1. We see that the first letter in the progression order denotes the parameter that varies the slowest through the bitstream, while the last letter denote the pa-

rameter that varies the fastest. The chosen progression order will vary with the area of application. E.g., processing time can be reduced by decoding only parts of a given codestream, where the obtained image will depend on the chosen progression order. It can be an image with reduced resolution, quality, number of components or region of support, compared to the original.

Listing 3.1: Pseudocode for LRCP progression order.

```
for l = 1 to ..
    for r = 1 to ..
        for c = 1 to ..
            for p = 1 to ..
                Add packet for layer l, resolution r, component c and position
                p to bitstream
```

3.1.4 Codestream Syntax

The JPEG2000 codestream can be generalized into two entities, namely header data describing the compressed bitstream and sections containing the actual compressed bitstream. The latter was discussed in section 3.1.3.

The basic building block for header data is the *marker segment*. This is a version of Key-Length-Value (KLV) coding. The general syntax can be seen in figure 3.8, and include three fields. The *Marker* is a two byte field always starting with 0xFF, and denotes the information contained within the marker segment. The *length* field is following the marker, and denotes the number of bytes in the marker segment, excluding the marker. The *marker segment parameters* is last, and is where the actual information is stored, for instance picture dimensions, subsampling and progression order.

The rules for a valid JPEG2000 Part 1 codestream are stated in Annex A in [14], and the reader is referred there for more details on valid codestream syntax.

3.1.5 Summary

JPEG2000 was not developed just to achieve higher compression efficiency than existing systems, but rather to produce an algorithm with a rich set of features within the same scheme and "address areas where current standards fail to produce the best quality or performance"[29]. The result can address a variety of existing and emerging compression applications and produces a highly flexible and embedded bitstream, with both lossless and lossy compression within the same integrated algorithm. Most of the credit for this should be given to two factors, namely the wavelet transform and entropy

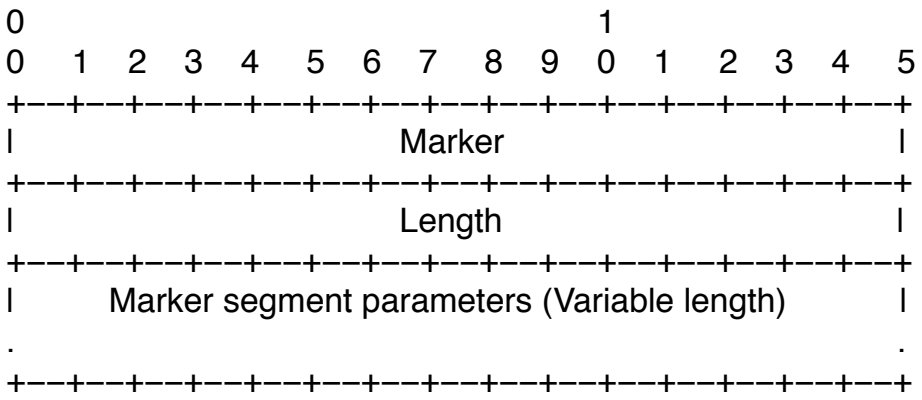


Figure 3.8: General codestream marker segment

coding. The wavelet transform is a flexible subband technic resulting in a fine partitioning. When combined with the EBCOT coding engine, powerful compression is obtained.

It should also be noted that JPEG2000 has many new important features not discussed here, e.g. Region of Interest (ROI) coding, error resilience, random access in codestream, metadata support and Digital Rights Management (DRM).

3.2 Video Coding

Image and video coding are closely related, and many of the same techniques are used. A video signal is basically a sequence of still images, where two dimensions represent the spacial video resolution and the third dimension represent the time line. In addition to the intraframe techniques described in this start of the chapter, video coding algorithms may apply interframe compression. I.e., intraframe compression is performed relative to information that is contained only within the current frame, while interframe compression in addition may utilize past and future frames.

Intraframe techniques aim at reducing the bit consumption within a given frame, and may choose from a number of decorrelating operations. Wavelet transform and fractal coding are used, but Discrete Cosines Transform (DCT) is most widely employed in modern video coders.

Since most video material meant for human consumptions contains smoothly moving objects, the change from one frame to another can be quite small. Huge bit rate savings can therefor be made by only coding the change from one frame to another. Motion estimation is another technique where frames are estimated as a best-match copy-paste from nearby sections in past and/or subsequent frames. Only vectors telling where to find the estimates are encoded together with the estimation error. This scheme is used

in the MPEG standards with variations. But interframes techniques have drawbacks. If errors are introduced or parts of a frame is lost, consecutive frames may not be correctly reconstructed. This error propagation between frames is the expense for the lowered bit rate. To bound the error propagation, frames are periodically encoded without interframe dependencies. Editing is also complicated by interframe coding, since changes made in one frame can impose changes in past or subsequent frames that depends on the current frame.

Although JPEG2000 is a still image compression standard, it is more than suitable for video coding. The DWT applied in JPEG2000 is superior in many ways to the classic DCT used in MPEG 2, MPEG 4 and VC-1, and the coding scheme does not suffer from the blocking artifacts seen in MPEG video. Editing is eased and errors do not propagate from frame to frame, because it is a intraframe coder. Multiple coding cycles are not needed to meet the target bit rate, since the bitstream is truncated after quantization and entropy coding. Because of the bitstream flexibility, a representation with reduced resolution and/or SNR may be extracted from an initially-encoded high-quality version without any transcoding. It should be noted that standards exist that obtain higher quality at low rates through exploiting inter-frame and inter-subband dependencies, but for high-rate applications JPEG2000 is a natural candidate as compression algorithm. The only evident drawbacks are that high computational complexity may compose a problem in real-time applications and that JPEG2000, being an image compression codec as opposed to video codec, typically has higher bandwidth requirements.

Chapter 4

General-Purpose Computing on GPUs

General-Purpose computing on Graphics Processing Units (GPGPU)[30] refers to techniques where calculations traditionally done by the Central Processing Unit (CPU), are handed over to the Graphics Processing Unit (GPU). Earlier, the GPU was used only to accelerate certain parts of the graphics pipeline, but now it can reduce the CPU load and/or increase the processing throughput for general purpose scientific and engineering computing. It is obvious that handing tasks over to the GPU can lower the CPU load, since the CPU has to do less work. Increased processing throughput is not equally trivial, but modern GPUs may have a core count in the range of eighth to several hundred, while CPUs seldom contains more than four. These cores can be utilized on none-graphics data through massive threaded parallelization if data dependencies between threads can be avoided, i.e. threads don't have to wait for results from other threads executing in parallel. Some calculations fit this scenario perfect, e.g. color space conversion where each pixel can be processed independently. In contrast are tasks that are truly serial i nature, e.g. decoding of a symbol sequence with variable symbol length. GPGPU will increase the performance of the first task considerable, while the latter will experience little or no improvement over the CPU benchmark. In general, computations with high arithmetic density map well to the GPU. Arithmetic density is defined as the ratio between the number of operations performed and the number of word transfered to and from memory. This is reasonable since the GPU cores can perform calculations much faster than they can access memory and have no cache.

Early attempts to use graphics hardware for general purpose computing required that the programmer had knowledge of the graphics pipeline, which is different in both terminology, data types, operations and programming model compared to other widely used programming paradigms. Thus, high level languages were created to hide graphics-related details from the

developer and include AMD Streams[31], CUDA[32] and OpenCL[33][1]. In the following, we will take a closer look at CUDA.

We get a historical outline for two important benchmarks for Intel CPUs and NVIDIA GPUs from figure 4.1, namely the number of FLoating point OPerations (FLOP) per second and memory bandwidth[34]. NVIDIA GPUs have gained a considerably advance on Intel CPUs over the last five years, judging by these benchmarks. Because the same instructions are executed on several data elements concurrently on a GPU, the requirements for sophisticated flow control and caching are lower and more transistors can be devoted to data processing than on CPUs. Some commentators have suggested that CPUs will be made superfluous, because of the superior performance found in modern GPUs. This is somewhat a misunderstanding, since GPUs lack important functionality to overtake the CPU, e.g. the ability to run an operating system.

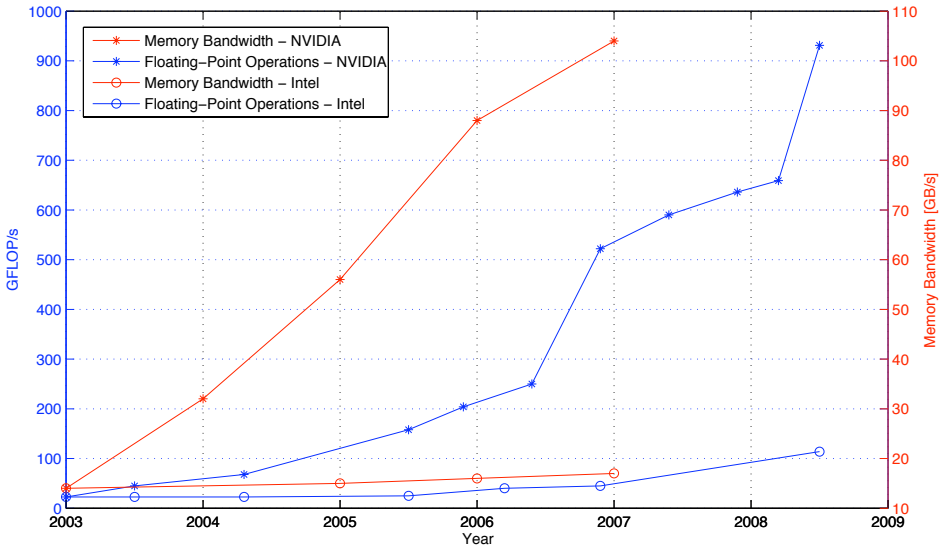


Figure 4.1: Peak floating-point operations per second and peak memory bandwidth for NVIDIA GPUs and Intel CPUs.

4.1 CUDA

In order to ease the usage of GPUs for programmers not familiar with the graphics pipeline, the CUDA[32] language was created by NVIDIA. It is an extension to the widely used C language, with a programming model easily

[1] OpenCL is a unified programming environment for a mix of multi-core CPUs, GPUs and other processors such as DSPs and Cell-type architectures.

understood by programmers already familiar with threaded applications. A throughout CUDA introduction is given in [34].

Two basic terms in the CUDA terminology are *host* and *device*, which are synonymous with CPU and GPU. Both entities manage its own DRAM, referred to as *host memory* and *device memory*. A simplified CUDA program is illustrated in figure 4.2 and is a mix of serial and parallel code. The serial code is executed on the host, while the parallel code is executed on the device. The first step in the CUDA related code is to transfer data from host memory to device memory. Then a special function is invoked on the device, which is named a *kernel*. Kernels have a special launch syntax:

```
kernelName<<<grid, block>>>(param1, param2, ..., paramN);
```

and are executed by multiple threads in parallel. Each tread processes different data and is extremely light weight, i.e. switching between threads is done with almost zero overhead. A kernel can be compared to host executed code placed inside a loop, but unlike the loop that processes data by running the code over and over in a consecutive way, a kernel is executed on several device cores concurrently[2]. Each device core processes different data elements. The result is transfered back from device memory to host memory when the kernel has finished.

In general, all data transfers between host and device, and all kernel invoking can be made either synchronous or asynchronous with respect to the host. Synchronous execution implies that the host will wait in a spinlock[3] or sleep until the invoked kernel has processed its data. On the other hand, asynchronous execution implies that the host invokes a kernel, performs other calculations while the kernel runs, and at some point finds out that the kernel has finished or waits until the kernel finishes. Both approaches have advantages and drawbacks. Asynchronous execution can increase the total amount of work done, but will add latency to the critical path if the host continues to perform long-lasting calculations after a critical kernel has finished. Synchronous execution reduces the latency, but inhibits the host from performing any useful work while the kernel executes. The best would if the host could perform some work while a kernel executed and return the instance the kernel finished, but this is hard to obtain since timing requirements will vary between platforms and data sets.

4.1.1 GPU Hardware and Execution Model

The graphics hardware and code execution model must be shortly visited, in order to understand how CUDA can attain high performance. NVIDIA

[2]The degree of concurrent execution will depend on the number of processor cores on the device.

[3]Loop where the thread simply waits (or "spins"), constantly checking a variable until it reaches a predefined value.

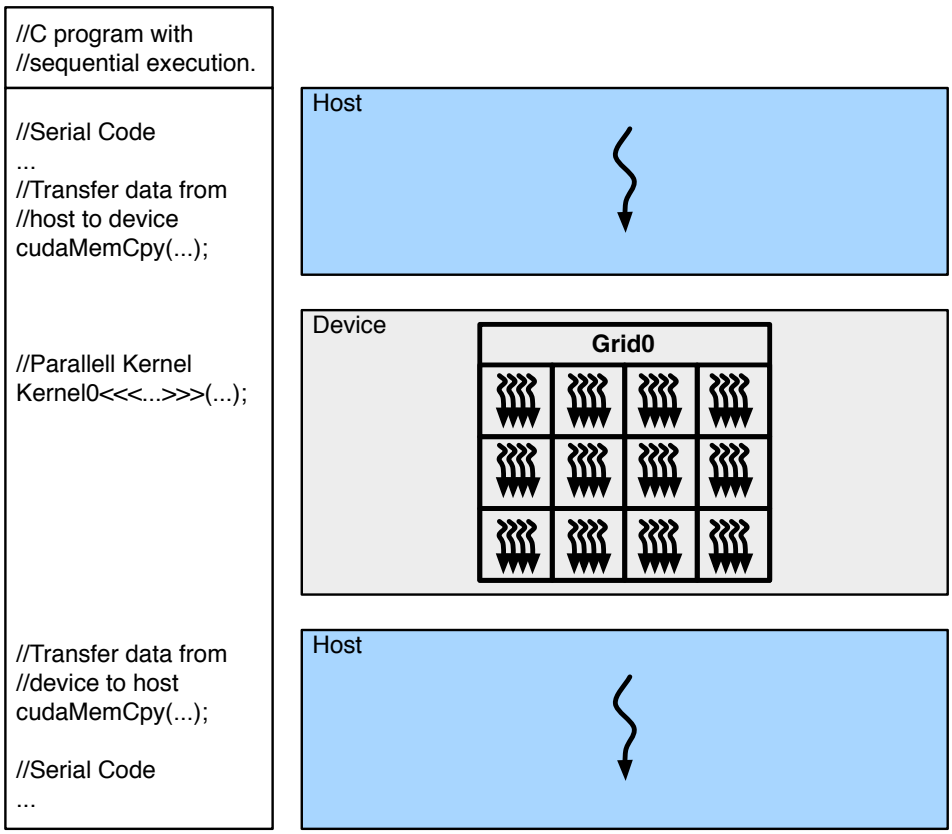


Figure 4.2: Synchronous CUDA program. Each curly arrow represent a worker thread.

GPUs consist of a scalable array of multithreaded Streaming Multiprocessors (SMs) and global memory accessible by all multiprocessors. The current generation of multiprocessors consists each of eight Scalar Processors(SP), on chip memory and a multithreaded instruction unit. Each multiprocessor employs a new architecture called SIMT (Single Instruction, Multiple Threads), where each multiprocessor maps the threads to scalar processors, and each thread executes independently with its own instruction address and register state. SIMT is in many ways equal to SIMD (Single Instruction, Multiple Data) used on CPUs, except for the important difference that it is hidden for the programmer in CUDA. Entry-level graphics cards can contain 8 or 16 SPs, while high-end workstations graphics cards can contain several hundred SPs.

Threads are organized into one, two or three dimensional structures called *thread blocks*, providing a natural way to process vectors, matrices and three dimensional arrays. A given thread block is executed on only one multiprocessor, establishing a way to synchronies and exchange information

between threads in the same thread block. Thread blocks are again organized in a two dimensional array called a *grid*, as seen in figure 4.3. The thread block size is a trade-off between efficient time-slicing within a block and multiprocessor occupancy, and typically has a value in the range 64-256. Allocating more threads per block is better for efficient time slicing between threads in the same block, but to large thread blocks will limit the obtained multiprocessor occupancy. High multiprocessor occupancy is attained when each multiprocessor has multiple active thread blocks at the same time. This hides memory latency and synchronization latency between thread blocks and implies that no scalar processor within the multiprocessor has to wait in an idle state. Given the thread block size, the grid size is determined by the total number of needed worker threads, which again is determined by the amount of data to process and how much data each tread processes.

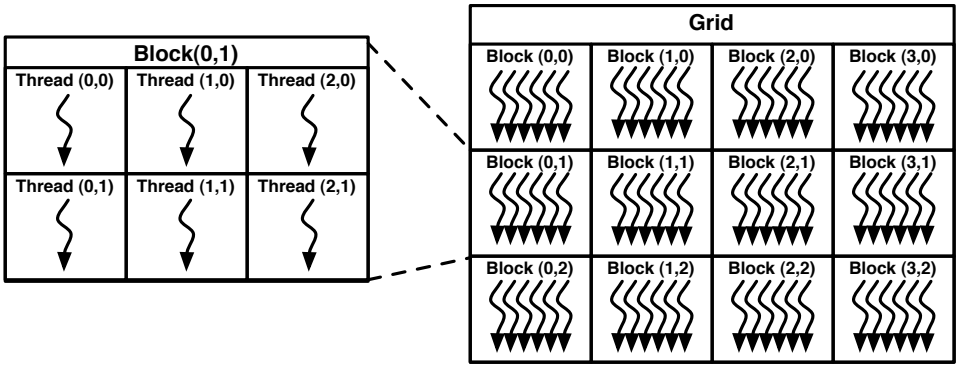


Figure 4.3: Thread hierarchy within the CUDA programming model.

Both block and grid dimensions are explicit specified by the programmer at kernel launch time, and together they govern how many elements that are processed. E.g, to process an image with dimensions 1024x1024, a block dimension of 16x16 treads and a grid dimension of 64x64 would be able to process the entire image if each tread processes one pixel. Each tread decides which data elements to process, based on embedded information regarding block- and grid size and indices within the grid and current block.

The scalability in CUDA lies in the number of thread blocks active at the same time, i.e. a device with many multiprocessors will be able to process many thread blocks concurrently, while a device with fewer multiprocessors has fewer active thread blocks. It should however be emphasized that a device with few multiprocessors is perfectly able to run the same program as one with more multiprocessors, it just take longer time. If a program has a large number of blocks per grid, it will experience performance scaling over future generations of GPUs. About 100 blocks per grid is advised for scaling to future devices, while 1000 will scale over several generations.

4.1.2 High Performance CUDA Code

In general, writing a program that processes its data in a correct manner is much easier than writing a program which in addition utilizes all available processing power in the underlying hardware. This is also true for CUDA, where any experiences programmer may write a CUDA program that processes its data correctly, but the true power within the GPU is only utilized if several considerations are taken into account.

First, the memory bandwidth between host and device is limited and often a bottleneck in both graphic and non-graphic applications. Off course, the device must be provided some initial data to process, but transfers between host and device should only be performed when absolute necessary. Further, a factor of two or more in transfer time may be saved, by allocating page-locked memory at the host. This ensures that the memory not changes physical addresses or is swapped out to disk by the operating system.

A similar performance increase can be expected if all threads access global memory in a smart fashion. More specific, if each thread accesses either a 32-, 64- or 128 bit words in memory in a consecutive manner and the memory is properly aligned, multiple memory accesses within a block will compile into one or two wide memory fetches. This technique is called *coalescing* and will give considerable speedups compared to non-coalesced memory access. Thus, coalesced memory access should always be the objective when planing the application architecture.

Another issue which that be considered, is to minimize the number of data dependent conditional branches made within a warp[4]. This is because each path taken within a warp is serially executed, until the paths converge, lowering the processing speed considerable. For further information on performance issues, consult chapter 5 in [34].

[4] A warp is a collection of 32 threads from the same block with increasing thread ID.

Chapter 5

Implementation

This chapter treats details regarding the chosen software architecture. The application was implemented in accordance with the software decoder specification given in section 2.2.

5.1 Architectural Overview

The architecture is illustrated in four parts, a *component diagram*, an *activity diagram*, a *class diagram* and of course the *source code*. The three first are based on the graphical notation techniques standardized in the Unified Modeling Language (UML). Each rectangular box in the component diagram, which can be seen in figure 5.1, represents an instance of the labeled class running as an individual thread[1]. The cylinders are storage elements and provide a mean for data exchange between threads. All buffers are implemented as FIFO queues protected by one or multiple mutexes. This ensures that no thread can read a buffer at the same time as another is writing to it. The activity diagram is given in appendix A and gives together with the class diagram in appendix B the schematic workflow of components in the system. The reader is advised to study the activity diagram and components diagram in order to understand the parallelization within the application.

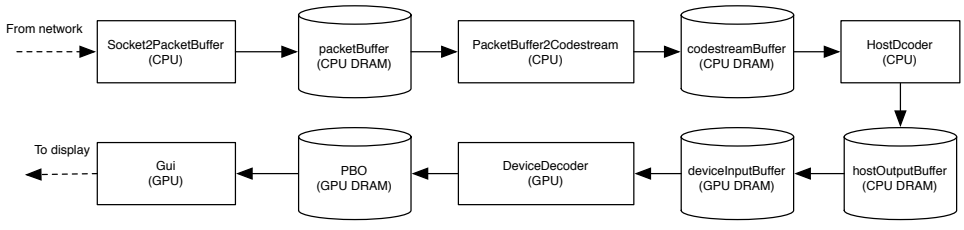


Figure 5.1: Component diagram for the software application.

[1]This is not true for the box named Gui and DeviceDecoder, which is executed by the same thread. HostDecoder is composed of two threads.

- The *Socket2PacketBuffer* class is responsible for joining the UDP multicast and receiving UDP packets. These UDP packets are then placed in a circular buffer named packetBuffer.
- *PacketBuffer2CodeStream* is a class that unites several UDP payloads from the packetBuffer into a single JPEG2000 encoded frame and places it in the codestreamBuffer.
- The *HostDecoder* class extracts a single JPEG2000 encoded frame from the codestreamBuffer, performs the CPU executed decoding and puts it in the hostOutputBuffer.
- The *DeviceDecoder* is responsible for transferring the partly decoded frame from CPU memory to GPU memory, and performing the final decoding on GPU. The result is written to a Pixel Buffer Object (PBO) residing in GPU memory.
- The *Gui* displays the data stored in the PBO after the DeviceDecoder has finished. This can be done with almost zero overhead, since the frame already reside in graphics memory.

5.2 Decoding Performance at Chosen Milestones

This section presents the decoder performance at chosen milestones during the development process. The results have been obtained by individually testing of the decoder elements, while all other application components were disabled, as described in section 5.5.1. This way the maximal performance for each element in the pipeline was revealed, and no synchronization was required between pipeline elements.

In order to estimate the theoretical gain obtained from mapping decoding steps from the CPU to the GPU, it would be advantageous to know the relative execution time for the different decoding steps in the JPEG2000 processing chain. However, this will vary considerably between different implementations, hardware platforms, image dimensions and codestream rates. The values reported in literature also vary over a wide interval. Thus, relative execution time can only be roughly estimated from [3, 35], which treats encoding with the 9/7 wavelet. It can be assumed that wavelet transform consume about 25-50%, EBOCT 35-65% , ICT 5-10% and quantization 5-10%. When instead considering decoding, EBCOT will have a smaller relative execution time, since forward EBCOT is more complex than reverse EBCOT. The best would be to perform an accurate run-time profiling[2] of the software and hardware in question, but this has been considered out off the scope for this thesis.

[2]Profiling is the use of a performance analysis tool that measures the behavior of a program as it executes, particularly the frequency and duration of function calls.

Table 5.1: Decoder performance for 1280x720 image at chosen milestones during the development process. Codestream read from local storage, at a rate of 45 Mbit/s.

Milestone	CPU decoder	GPU decoder	Increase from baseline
JasPer	1fps	*	-
Kakadu:			
baseline	27.3fps	-	1
enhancement 1	43.4fps	312fps	1.59
enhancement 2	50.5fps	468fps	1.85
enhancement 3	56.7fps	158fps**	2.08

*Unknown.

**IDWT not included.

5.2.1 JasPer

JasPer[36] is an open-source initiative to provide a free software-based reference implementation of JPEG2000, based on CPU executing alone. Initial tests were done with a GPU accelerated version of JasPer, enhanced with forward and inverse DWT on GPU[12]. Unsatisfactory performance made this codec uninteresting for further investigations.

5.2.2 Kakadu, Baseline

Kakadu[37] is a commercial JPEG2000 SDK, which also is based on CPU executing alone. This milestone represents the baseline performance, with all decoding done on CPU. It should be noted that the ICT routine used here, not had been optimized. Thus, higher throughput can be expected based on CPU decoding alone, with an optimized ICT routine.

5.2.3 Kakadu, Enhancement 1

This milestone represents the first version employing GPU assisted decoding. More specific, chroma upsampling and ICT was mapped from the CPU to the GPU. This gave a fair throughput improvement over the baseline decoder.

5.2.4 Kakadu, Enhancement 2

Thread switching latency, which is the time needed by the operating system to switch the CPU to another thread, can accumulate to significant amounts if the switching frequency is high. By forcing the decoder threads to utilize their allocated time slice better, less thread switching is performed and higher throughput is obtained. I.e. by explicitly telling each decoder thread to perform a bigger amount of work before yielding the rest of its time slice

to other waiting threads, more of the available processing power in the CPU are exploited. This came at the expense of increased memory consumption, but was no problem since the application has a small memory footprint.

The decoding steps are allocated between CPU and GPU as in enhancement 1. The GPU performance was enhanced by performing less uncoalesced memory access.

5.2.5 Kakadu, Enhancement 3

In order to obtain a more balanced pipeline, an attempt was made to map the the reverse DWT from the CPU to the GPU. This included three decoder changes.

First, the internal IDWT in the Kakadu core system had to be bypassed, which proved to be a demanding task. The Kakadu core system is pipelined and optimized for high throughput, and therefor hard to understand. Instead of changing the source code, the internal IDWT was bypassed by adding an Arbitrary Transform Kernel (ATK) marker segment to the JPEG2000 codestream prior to CPU executed decoding. An arbitrary transform kernel is a part 2 extension, usually employed when encoding with wavelets other than default CDF 9/7 or 5/3. But rather than encoding with a special wavelet, the decoder is explicit told to use a special wavelet called the "lazy-wavelet". This is simply a lifting process with zero lifting steps, or mathematically equivalent, all lifting coefficients set to zero. Thus, output from the CPU executed decoder is now DWT coefficients, since the decoder apply an IDWT with no effect. From a performance perspective this is believed to be suboptimal. Although the throughput increased with about 12% for the CPU based decoder element, a further increase in throughput is expected if the internal IDWT in the Kakadu core system is bypassed with a source code change. This is based on the assumption that IDWT consumes a larger part of the total decoding time for the CPU executed decoder, as discussed in the start of section 5.2.

Second, since the wavelet coefficients are interleaved as illustrated in figure 3.6, they had to be deinterleaved before the reverse transform. This was done with a CUDA kernel. However, because the deinterleaving involves accessing memory in a unordered fashion, the kernel has few coalesced memory accesses and the resulting performance is weak.

Last, an attempt to perform IDWT on the GPU was made. Initially, the author wished to write a CUDA kernel performing reverse 9/7 wavelet transform, but limited time made this impossible. A CUDA kernel therefor had to be obtained from other sources. Attempts were first made with a CUDA kernel from the Schroedinger project[38], but it proved incapable of performing a correct reverse transform. A second attempt was made with

a CUDA kernel provided by the courtesy of Jiri Matela[3]. However, this code contained approximations that introduced unacceptable noise in the decoded image. Therefor, no successful decoding was obtained with IDWT mapped to the GPU.

5.3 Core Components Revisited

The components briefly visited in section 5.1 are now discussed more thorough. For the deepest level of details, the reader is referred to the source code given in appendix E.

5.3.1 Socet2PacketBuffer: Receive UDP Packets

The transport layer services are maintained by the *User Datagram Protocol* (UDP). A datagram socket must be created in order to join a UDP multicast and thereby receive UDP packets. This is an interface between an application process and the IP protocol stack provided by the operating system. The operation system uniquely identifies a datagram socket by the protocol, the local IP address and port number. The operating system forwards incoming IP data packets to the correct application by extracting the above socket address information from the IP and UDP headers. Sockets were developed at Berkley as an abstraction to enable computer communication over different mediums, and forms the de-facto standard for networking. Windows Sockets 2 builds on the Berkley Sockets and provides the necessary methods to join a multicast and receive UDP packets. The received UDP payloads are placed in a circular buffer to facilitate detection of packet reordering, duplication, deletion and immunity to mild jitter. The application layer services are maintained by the *Real-time Transport Protocol* (RTP), where the RTP header contains a sequence number used to detect packet reordering, insertion and deletion.

5.3.2 PacketBuffer2Codestream: Reconstruct JPEG2000 Codestream from Multiple UDP Payloads

Figure 5.2 shows an example of a packetized codestream originating from a TVG430. As seen in this figure, each UDP payload is composed of a RTP header and a RTP payload. The RTP payload is composed of a JPEG2000 codestream segments, encapsulated in the Material eXchange Format (MXF)[39, 40]. MXF is a container format used to carry sound-, image- and metadata content, collectively termed essence. It is also used for synchronization. Since each compressed video frame is divided amongst multiple UDP payloads, several codestream segments must be extracted and put together in order to obtain a complete frame ready for decoding.

[3]Master student at the Masaryk University, Czech Republic.

The compressed video frames are coded with the YCbCr color space, where luma and chroma components are received as separate JPEG2000 codestreams, i.e., the Y component (luma) is encoded as one codestream, while the Cb and Cr components (chroma) are encoded as another codestream. These two codestreams could be decoded separately and combined before the YCbCr to RGB color space transform, but instead they are combined into one single codestream before decoding in order to avoid multiple decoders. This is possible without any transcoding, owing to the flexibility in JPEG2000. Some minor changes must be made in the codestream header, since the luma and chroma codestreams contains redundant marker segments[4], but no changes are made to the compressed bitstreams. By inserting a Progression Order Change (POC) marker segment in the codestream header and encapsulating the luma and chroma codestream in individual tile-parts, the underlying decoder is able to correctly decode all three components in one run. The meaning of progression order within JPEG2000 is explained in section 3.1.3. For more information on the POC marker segment and the changes made in the codestream header, please consult annex A in [14] and the source code in appendix E, respectively.

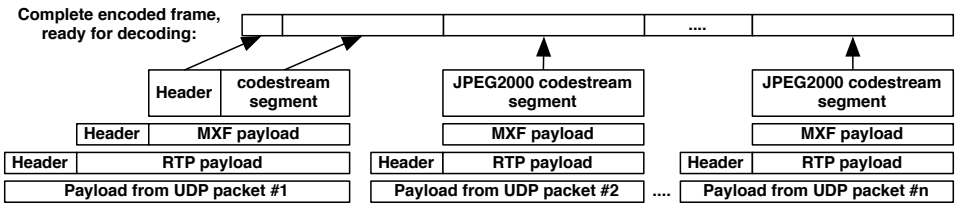


Figure 5.2: Simplified overview of the TVG430 bitstream.

5.3.3 HostDecoder and DeviceDecoder: Decoding of JPEG2000 Codestream

The implemented decoder is composed of two parts, where the first is named HostDecoder[5], is executed on the CPU and involve all decoding steps up to and including inverse wavelet transform. The second part is named DeviceDecoder[5], include chroma upsampling and Irreversible Color Transform (ICT) and is executed on the GPU. This architectureis identical to the one described in section 5.2.4, named Kakadu enhancement 2.

HostDecoder is based on Kakadu[37], which is a JPEG2000 SDK and a complete implementation of JPEG2000 Part 1, and a great deal of Parts 2 and 3. More specific, the CPU executed part of the decoder is based on the high-level application kdu_stripe_decompressor, which again is based on the low-level application kdu_decoder, both found in the Kakadu SDK.

[4]See section 3.1.4 for more information on marker segments.

[5]The names are motivated by the CUDA naming convention.

Input to the decoder is a JPEG2000 compressed codestream, while the output is a three component bitmap represented with the YCbCr color space with 4:2:2 subsampling.

The CUDA language was used to perform all GPU processing in *DeviceDecoder*. After the output from the CPU executed decoder is transfered from host to device memory, a Pixel Buffer Object (PBO) is mapped into CUDA memory space. This enables the computed RGBA[6] values with 4:4:4 subsampling to be directly written to the PBO during the concurrent chroma upsampling and Irreversible Color Transform (ICT). The ICT is from the YCbCr color space to the RGB color space, and is treated mathematically in appendix C. The PBO is used in order to reduce the overhead connected with the later displaying of the result, which is further discussed in section 5.3.4.

The CPU and GPU are pipelined to exploit the inherent parallelism between them, meaning that both the CPU an GPU can perform decoding at the same time, just on different frames. Synchronization is needed when accessing shared buffers, and is obtained with mutexes. The synchronization is based on the assumption that the CPU decoder is slower than the GPU decoder, meaning that frames will be lost if the opposite is true. This is a tradeoff between fool-prof synchronization and execution time, since additional synchronization would add latency to the critical path. The buffer between the CPU and GPU executed decoder should probably be increased to accommodate multiple partly decoded frames, in order to better absorb the inevitable jitter in processing time. Currently it holds only one partly decoded frame.

In addition to the implemented decoder described above, a serious attempt has been made to map the inverse wavelet transform from CPU to GPU. This is explained more thorough in section 5.2.5.

5.3.4 Gui: Application Front-End

The application is implemented with a graphical user interface, which also is composed of two parts. The first part is a startup wizard based on the Qt framework[41], used at startup to obtain the multicast address and port number for the UDP multicast. This window can be seen in figure 5.3.

The second part is the main window used to display the decoded video frames, and can be seen in figure 5.4. It is implemented with the OpenGL interoperability found in the CUDA runtime API, which enables the Gui to create a texture based on the above discussed PBO. This texture is then drawn to the screen. As mentioned in section 4.1.2, transfers between host and device are costly with respect to execution time. This architecture completely avoids data read-back from device to host, since the host does not access the data after they are transfered to device memory. This is accom-

[6] All alpha values are set fully opaque.

plished since the decoded stream is meant for immediate visual consumption, and CUDA facilitate a simple binding to OpenGL.

Because of restrictions in CUDA with respect to threads and memory access, the GPU decoding procedure described in section 5.3.3 has to be called from the Gui main loop. The reason is that "any CUDA resources created through the runtime in one host thread cannot be used by the runtime from another host thread"[7]. This means that two host threads not can use the same device memory. The same restrictions seems to apply to a PBO, because the PBO has not been successfully mapped to two threads for writing or reading. The only known solution is to let the thread running the Gui main-loop copy data from hostOutputBuffer to deviceInputBuffer, perform the GPU processing and use the OpenGL bindings to display the decoded frame. This may be a little illogical, since the GUI not should be involved in the decoding process, but it is the solution used in the OpenGL sample applications in the CUDA SDK. A more logical partitioning between decoder and GUI would be obtained, if the decoder could update the PBO from a thread other than the GUI main-loop thread. However, the current arrangement is advantageous from a performance perspective, since it provide a simple way to pipeline the CPU and GPU decoding operations.

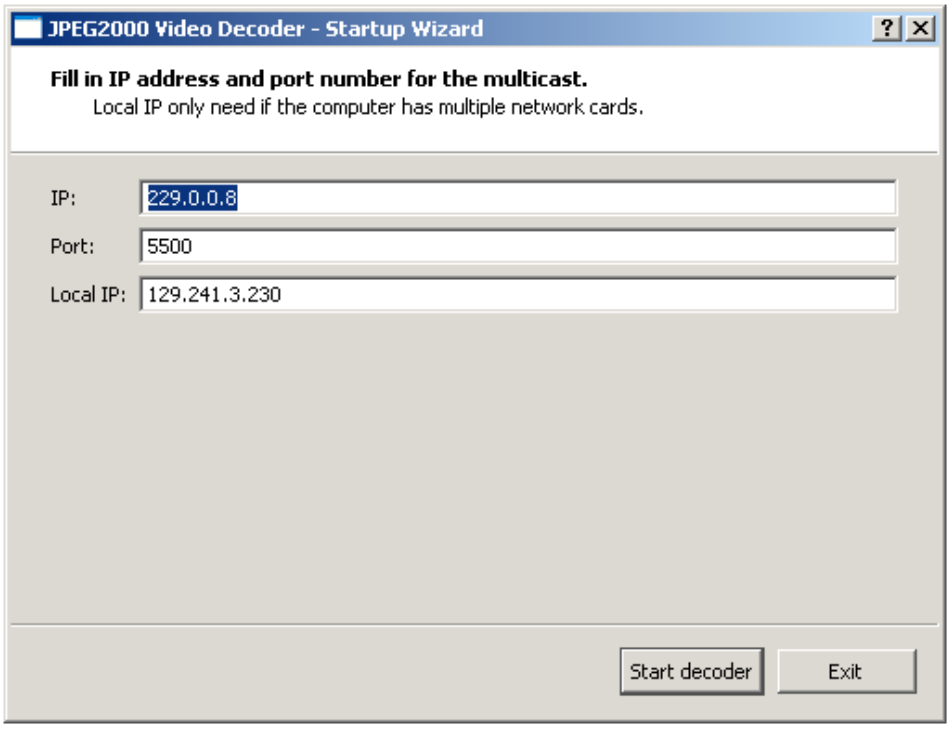


Figure 5.3: Screenshot of application startup wizard.

[7]See [34] section 4.5.1.1.

Figure 5.4: Screenshot of application main window.

5.4 Software and Hardware Tools

The application has been developed on Windows XP SP3, although the only platform specific code are the socket handling and synchronization between treads. Visual Studio Professional v9.0.21022.8 has been the deployed IDE, together with the Intel C++ Compiler v11.0.072. Kakadu v6.1.1 was used for all CPU executed decoding. All CUDA related code has been compiled with the MS Visual C++ compiler and the NVCC compiler included in the CUDA 2.1 SDK. CUDA Notebook driver 185.85 has been used when performing the GPU calculations. All performance results have been obtained on a HP Elitebook 8730w with Intel Core 2 Duo T9600 (2.80 GHz) CPU, NVIDIA Quadro FX 2700M[8] GPU and 4GB RAM.

5.5 Test Setup

The application has been tested both during and after the development process. This section presents two different test setups, where the first tests only the decoder pipeline, while the second tests all application components. Sequences with raw video material have been obtained from Swedish television (SVT) via the European Broadcasting Union[9]. The videos were represented as uncompressed individual frames in the SGI format, with a resolution of 1280x720.

[8] With 512 MB DRAM and 48 cores(SPs)
[9] ftp: ftp.ebu.ch, username: hdseqs, password: 4testing

5.5.1 Decoding from Local Memory

In this setup, various JPEG2000 codestreams are read from file to CPU memory and then decoded over and over within a loop. Both the Socket2PacketBuffer, PacketBuffer2Codestream and Gui module are deactivated. This setup focuses purely on decoding performance and provides an upper limit on the attainable decoding rates on the given hardware, since all processing power are allocated to the decoding process. The decoder pipeline elements can be tested both individually and together, More specific, they can run in a consecutive manner or concurrently.

5.5.2 Decoding from IP Interface

In this setup, all application modules are concurrently active. I.e. UDP packets are received through the IP interface, compressed frames reconstructed from multiple UDP payloads, and frames are decoded and displayed. The test sequences from SVT were fed from a ClipRecorder[42] at 50 fps through HD-SDI to a T-VIPS TVG430 video gateway for JPEG2000 encoding and IP encapsulation. ClipRecorder is a device for disk-based recording and playout, capable of uncompressed output through HD-SDI. The bitstream is after encoding transmitted via an IP router to a laptop with the decoder software. The entire test setup can be seen in figure 5.5 and in appendix D.

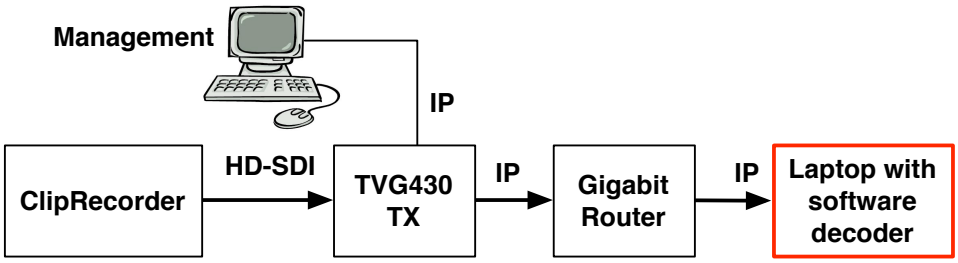


Figure 5.5: Overview of the test setup.

Chapter 6

Results

This chapter presents the achieved frame rates for the finalized application. The decoder performance has been measured in four way, which differ in the way the compressed codestream is obtain and the number of active modules within the application. The tested architecture was Kakadu enhancement 2, as described in section 5.2.4.

First, each element in the decoder pipeline was tested individually as described in section 5.5.1, to identify whether the obtained pipeline was balanced or not. Compressed video frames were read from file prior to decoding. The attained frame rates for various codestream rates are plotted in green in figure 6.1, with triangles and stars at the data points for the GPU and CPU executed parts of the decoder respectively. Note the discontinuous x-axis.

The second setup also tested the decoder pipeline alone. But now both pipeline elements were tested together, by running them concurrently on the CPU and GPU. The resulting frame rate is plotted in red in figure 6.1, with squares at the data points. The target frame rate was in section 2.2 specified to 50 frames per second. From figure 6.1 it can be seen that this is achieved for codestream rates below 45 Mbit/s.

Last, the entire application was tested by activating all modules as described in section 5.5.2. I.e. UDP packets were received through the IP interface, compressed frames reconstructed from multiple UDP payloads, and frames decoded and displayed. The attained frame rate for various codestream rates is plotted in blue in figure 6.1, with circles at the data points. The total received data rate was about 10% higher than the received codestream rate, due to MXF, RTP and IP encapsulation of the compressed video stream.

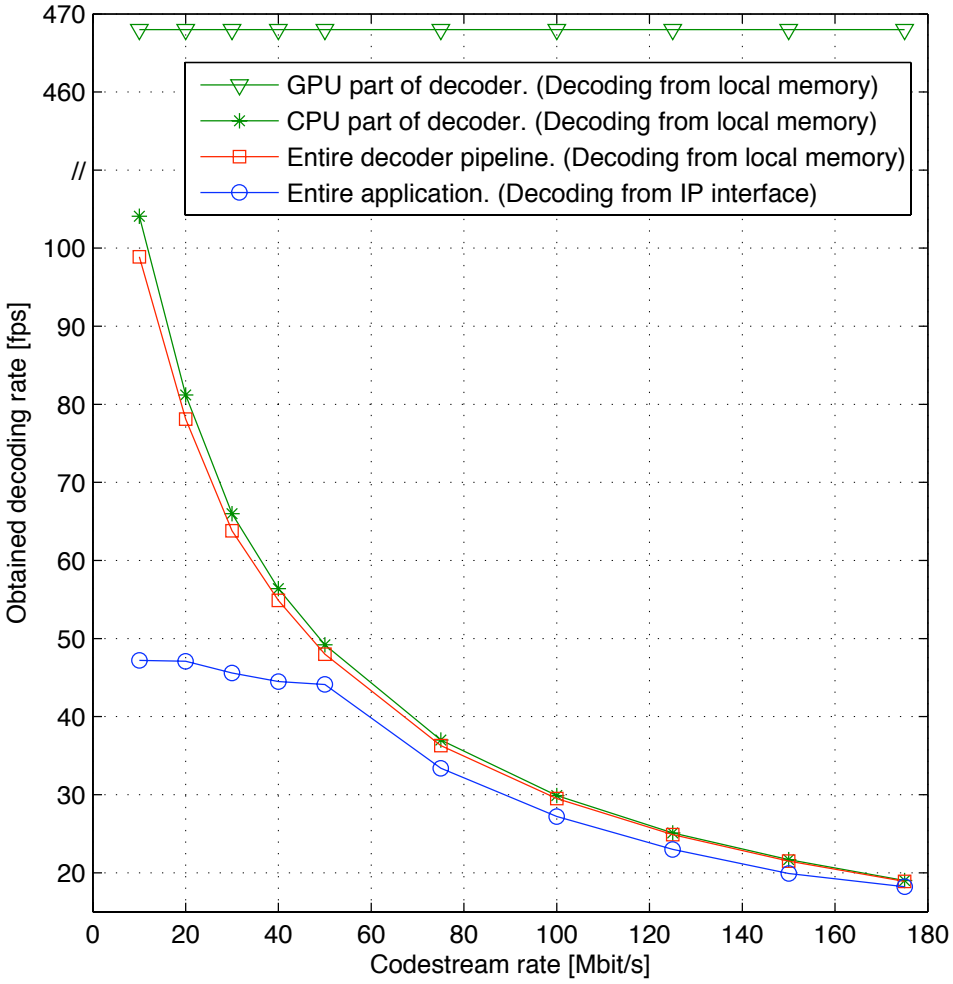


Figure 6.1: Obtained frame rate for the finalized application. Note the discontinuous x-axis.

Chapter 7

Discussion

The results presented in chapter 6 have been obtained by pipelining the CPU and GPU within the JPEG2000 decoder. I.e. the decoding steps have been divided between the CPU and the GPU, and performed concurrently. This parallelization is simplified by the fact that the video stream is inter-frame encoded only, and thereby without any dependencies between frames. Since all frames must propagate through the entire pipeline before they can be displayed, the entire decoder is only as fast as the slowest element in the pipeline. Maximal throughput is obtained when both elements in the pipeline have equal processing time, i.e., the optimal pipeline is perfectly balanced. This can be proven by a simple contradiction. Assume that all elements initially have equal processing time, and that one decoding task is moved from one pipeline element to another with the intention to increase the overall performance. Since the element receiving the additional decoding task must perform more calculations than before, its processing time must indisputably increase. Because the slowest element determines the pipeline throughput, overall performance will decrease.

From figure 6.1, it is evident that the performance for the entire decoder pipeline is marginally lower than the performance for the CPU part of the decoder. This confirms the statement that the pipeline throughput is limited by the slowest element in the pipeline. The difference between the CPU part of the decoder and the entire pipeline, is caused by the time consuming thread-safe synchronization between the two pipeline elements. Despite that the decoder pipeline is below 50 fps for codestream rates of 45 Mbit/s or higher, real-time decoding can be obtained at these rates. Owing to the JPEG2000 flexibility and employed progression order, it is possible to decode a truncated version of the received bitstream. This will reduce the quality, compared to full rate decoding, but can yield the desired frame rate.

From figure 6.1, it is also obvious that the application never obtains the target frame rate of 50 fps, when receiving the video stream through the IP interface. At low rates, the application is limited to about 46 fps, due

to packet loss at the socket level. This packet loss renders it impossible to correctly reconstruct all JPEG2000 codestreams, since each compressed frame is composed of several UDP payloads. The reason behind the packet loss has not been determined, but it is most likely caused by high bit rate combined with high CPU load. I.e. a long inactive period for the socket thread will result in packet loss, since network card buffering is limited. A codestream rate of 40 Mbit/s equals 5000 UDP packets each second[1], where one lost packet results in one incomplete frame. If this packet loss could be avoided, real-time decoding should be obtained at codestream rates up to about 40 Mbit/s, when receiving the bitstream through the IP interface. At high codestream rates, the performance for the entire application is limited by the decoder performance, and not the packet loss. The difference between the frame rates obtained with the decoder pipeline only and the entire application, is caused by the overhead connected to reception and buffering of UPD packets, reconstruction of the encoded frames from multiple UDP payloads and display of the decoded video.

Although the developed decoder is applied to a wide range of codestream rates, not all decoding steps have varying execution time. More specific, reverse EBCOT has varying execution time, while chroma upsampling and irreversible color transform have execution times that are invariant to the compressed codestream rate. This is confirmed by the results plotted in figure 6.1 where the GPU part of the decoder has constant execution time, while the CPU part has an execution time that monotonically decreases with the codestream rate. This provide important knowledge, when designing a system for a given frame rate, video resolution or codestream rate. E.g. lowering the codestream rate will not improve performance for the GPU based decoder. Similarly, higher spatial video resolution does undeniably imply higher load at the GPU side, since the problem size is directly determined by the spatial video resolution.

Because estimating the execution time for a given decoding step on either CPU or GPU is complex, the pipeline balancing process has been rather practical during the work with this thesis. More specific, the idea has been to map one decoding step at the time from CPU to GPU, and measure the execution time for each pipeline element afterwards. Partly mapping a decoding step has been considered to complex. Luma upsampling and color space conversion were first mapped from the CPU to the GPU, which resulted in a highly unbalanced pipeline. Consequently, it was attempted to map reverse wavelet transform from the CPU to the GPU. This did not work out due to the lack of a IDWT routine for the GPU, as described in section 5.2.5. However, based on the performance obtained with Kakadu enhancement 3 and performance reports from the author of the second tested CUDA IDWT code, is should be possible to perform IDWT on the GPU

[1]Each UDP payload carries about 1kB of JPEG2000 data.

while staying above 50 fps for the GPU pipeline element. More specific, one level IDWT has been reported to take 0.386 ms[2] on a 480 core GPU. If the IDWT execution time can be assumed to be proportional to the number of GPU cores and the image dimensions, a 5 level IDWT on three components with 4:2:2 subsampling on a 48 core GPU, should be possible in approximately 10 ms. This would give a significant speedup and is a natural extension to the current architecture.

[2]Image resolution of 1280x720, 8 bpp.

Chapter 8

Conclusion

A complete playback application for JPEG2000 video has been developed within the scope of this thesis, where most of the attention has been given to the decoder design and implementation. The application shows that real-time decoding of a HD JPEG2000 video stream is feasible on general-purpose computer hardware and it is targeting situations where a special-purpose hardware decoder is unsuitable. The application obtains real-time playback of a 720p video stream at codestream rates up to 40 Mbit/s. However, the maximal obtained frame rate is limited to 46 fps, due to packet loss at the socket-level.

Real-time demands have been met by including the GPU in the decoder pipeline and mapping tasks from the CPU to the GPU. More specific, the CPU performed all decoding steps up to and including reverse wavelet transform, while chroma upsampling and color space conversion were performed on the GPU. The obtained pipeline provided increased throughput for playback applications, where the video content is meant for instantaneous visual consumption, compared to CPU decoding alone. The GPU has proven to be a powerful device usable also on none-graphics data, and the use of GPUs in none-graphics processing will most likely increase in the future.

Although considerable effort has been put into finding an architecture that utilize the available processing power efficiently, further optimization will always be possible. The obtained decoder pipeline is highly unbalanced, and better performance is expected if inverse wavelet transform is successfully mapped from the CPU to the GPU.

Chapter 9

Future Work

This chapter briefly presents work that should be carried out in order to make the application more sophisticated and stable.

First, measures should be made to enhance the socket layer of the application, since it currently is experiencing an undesirable packet loss. Methods for packet reordering and insertion has not been implemented since all tests have been performed on an IP network with only one hop between encoder and decoder, but this should also be included in future versions.

Secondly, the employed video gateway has functionality not supported by the developed software decoder. This include scrambling, forward error correction and uncompressed audio transmission. A decoder supporting this functionality will increase its market value.

References

[1] D. Santa-Cruza, T. Ebrahimia, J. Askelof, M. Larssonb, and C. A. Christopoulosb, "JPEG2000 Still Image Coding Versus Other Standards," *Applications of Digital Image Processing XXIII*, vol. 4115, pp. 446–454, 2000.

[2] M. Horowitz, A. Joch, F. Kossenthi, and A. Hallantiro, "H.264 Baseline Profile Decoder Complexity Analysis," *IEEE Transactions on Video Technology*, vol. 13, pp. 704–716, 2003.

[3] C. Garcia, C. Tenllado, L. Pinuel, and M. Prieto, "JPEG2000 Optimization in General Purpose Microprocessors," 2005.

[4] T. Moriyoshi, H. Shinohara, T. Miyazaki, and I. Kuroda, "Real-Time Software Video Codec With a Fast Adaptive Motion Vector Search," *IEEE*, 1999.

[5] V. Iverson, J. McVeigh, and B. Reese, "Real-Time H.264/Avc Codec on Intel Architectures," *International Conference on Image Processing*, 2004.

[6] P. W. Gang Ren and D. Padua, "An Empirical Study on the Vectorization of Multimedia Applications for Multimedia Extensions," *In Proc. of the 19th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS05)*, 2005.

[7] C. J. Thompson, S. Hahn, and M. Oskin, "Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis," *Proc. ACM/IEEE MICRO-35*, pp. 306–317, 2002.

[8] E. S. Larsen and D. McAllister, "Fast Matrix Multiplies using Graphics Hardware," *Proc. IEEE Supercomputing.*, p. 55, 2001.

[9] B. Fang, G. Shen, S. Li, and H. Chen, "Techniques for Efficient DCT/IDCT Implementation on Generic GPU,"

[10] M. Hopf and T. Ertl, "Hardware Accelerated Wavelet Transformations," *In Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym 00*, pp. 93–103, 2000.

[11] K. Moreland and E. Angel, "The FFT on a GPU," *Proc. SIGGRAPH*, pp. 112–136, 2003.

[12] T.-T. Wong, C.-S. Leung, P.-A. Heng, and J. Wang, "Discrete Wavelet Transform on Consumer-Level Graphics Hardware," *IEEE Transactions on Multimedia*, vol. 9, pp. 668–673, 2007.

[13] G. Shen, G.-P. Gao, S. Li, H.-Y. Shum, and Y.-Q. Zhang, "Accelerate Video Decoding With Generic GPU," *IEEE Transactions on Circuits and Systems for Video Technology, Vol. 15, No. 5*, pp. 685–693, 2005.

[14] ISO/IEC, *15444-1 - JPEG 2000 Image Coding System: Core Coding System*, 2004.

[15] Digital Cinema Initiatives, *Digital Cinema System Specification v 1.2*, 2008.

[16] *T-VIPS TVG430 HD JPEG2000*. http://www.t-vips.com.

[17] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, pp. 337–343, 1977.

[18] D. Taubman and M. Marcellin, *JPEG2000: Image compression fundamentals, standards and practice*. Kluwer Academic Publishers, 2001.

[19] M. Rabbani and R. Joshi, "An Overview of the JPEG2000 Still Image Compression Standard," *Signal Processing: Image Communication*, vol. 17, pp. 3–48, 2002.

[20] A. Skodras, C. Christopoulos, and T. Ebrahimi, "The JPEG2000 Still Image Compression Standard," *IEEE Signal Processing Magazine*, vol. 18, September 2001.

[21] W. Sweldens, "The Lifting Scheme: A Construction of Second Generation Wavelets," *J. Math Analysis vol. 29*, pp. 511–546, 1998.

[22] I. Daubechies and W. Sweldens, "Factoring Wavelet Transforms Into Lifting Steps," *J. Fourier Anal. Appl. 4*, pp. 247–269, 1998.

[23] S. Gnavi, B. Penna, M. Grangetto, E. Magli, and G. Olmo, "Wavelet Kernels on a DSP: A Comparison Between Lifting and Filter Banks for Image Coding," *Applied signal processing.*, pp. 981–989, 2002.

[24] K. A. Kotteri, S. Barua, A. E. Bell, and J. E. Carletta, "A Comparison of Hardware Implementations of the Biorthogonal 9/7 DWT: Convolution Versus Lifting," *Ieee Transactions on Circuits and Systems: Express Briefs, Vol. 52.*, pp. 256–260, 2005.

[25] C. Tenllado, J. Setoain, M. Prieto, L. Pinuel, and F. Tirado, "Parallel Implementation of the 2D Discrete Wavelet Transform on Graphics Processing Units: Filter Bank versus Lifting," *IEEE Transactions on Parallel and Distributed Systems, Vol. 19, No. 3*, pp. 299–310, 2008.

[26] G. Sullivan, "Efficient Scalar Quantization of Exponential and Laplacian Variables," *IEEE Trans. Inform.*, vol. 42, pp. 1365–1374, 1996.

[27] D. S. Taubman, "High Performance Scalable Image Compression With EBCOT," *IEEE Trans. Image Proc.*, vol. 9, 2000.

[28] W. Pennebaker, J. Mitchell, G. Langdon, and R. Arps, "An Overview of the Basic Principles of the Q-Coder Adaptive Binary Arithmetic Coder," *IBM J. Res. Development*, vol. 16, pp. 717–726, 1988.

[29] "Call for contributions for JPEG2000 (JTC 1.29.14, 15444): image coding system," *ISO/IEC JTC1/SC29/WG1 N505*, 1997.

[30] *General-Purpose Computation on Graphics Hardware.* www.gpgpu.org.

[31] *AMD Stream Technology.* http://ati.amd.com/technology/streamcomputing/.

[32] *CUDA.* http://www.nvidia.com/cuda.

[33] *OpenCL.* http://www.khronos.org/opencl/.

[34] *Cuda Progamming Guide V2.1.* http://www.nvidia.com/cuda.

[35] L.-G. Chen, C.-J. Lian, K.-F. Chen, and H.-H. Chen, "Analysis and Architecture Design of JPEG2000," *IEEE International Conference on Multimedia and Expo*, 2001.

[36] M. D. Adams and F. Kossentini, "Jasper: a Software-Based JPEG2000 Codec Implementation," *IEEE Transactions on Multimedia*, vol. 9, pp. 668–673, 2007.

[37] *Kakadu JPEG2000 SDK, written by Prof. David Taubman.* http://www.kakadusoftware.com/.

[38] *Dirac Video Compression.* http://www.diracvideo.org/.

[39] SMPTE, *Material Exchange Format (MXF) - File Format Specification, S377m*, 2004. http://www.smpte.org.

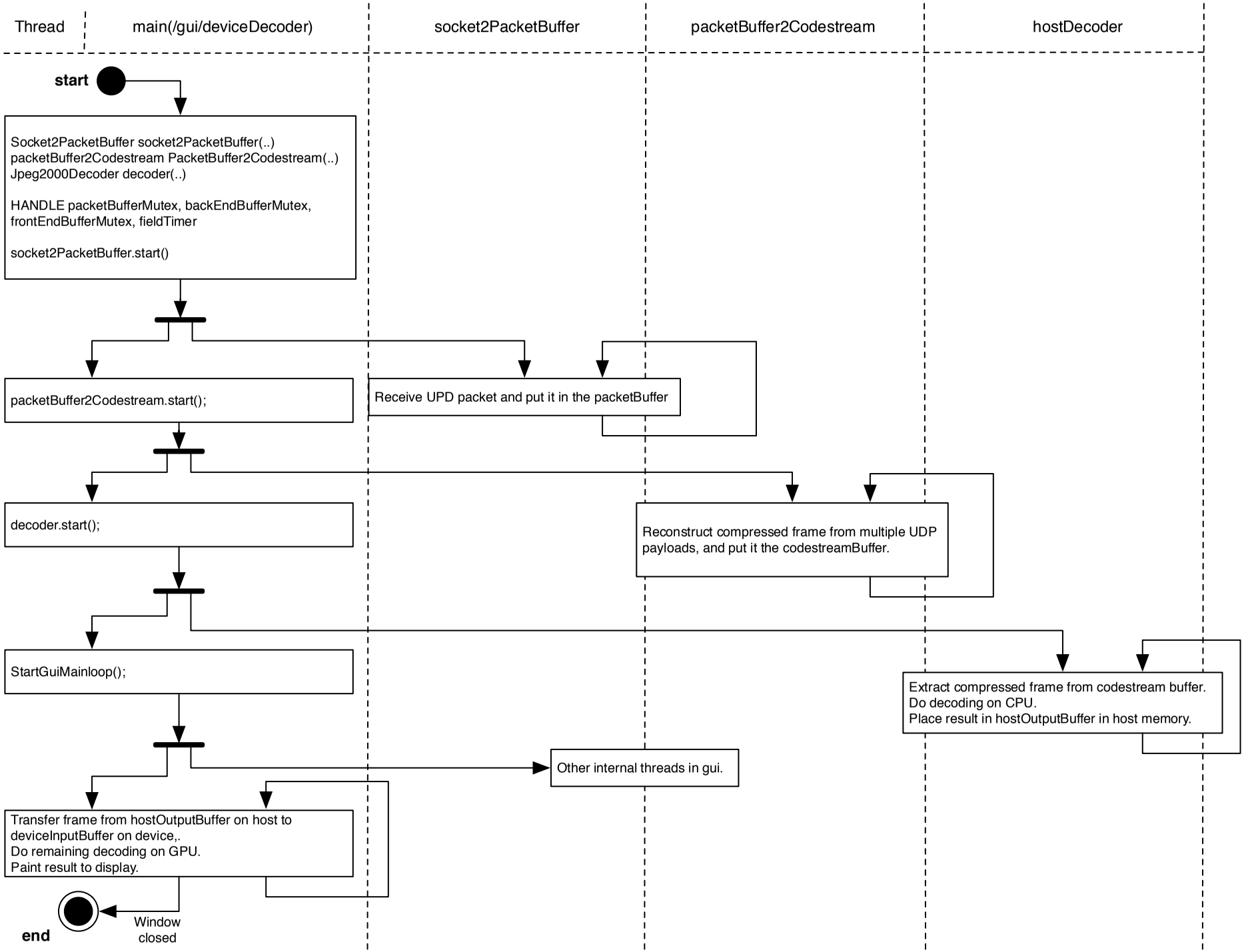
[40] SMPTE, *Material Exchange Format (MXF) - MXF Generic Container, S379m*, 2004. http://www.smpte.org.

[41] *Qt Cross-Platform Application Framework.* http://www.qtsoftware.com/products.

[42] *DigitalVideo Computing.* www.digitalcomputing.de.

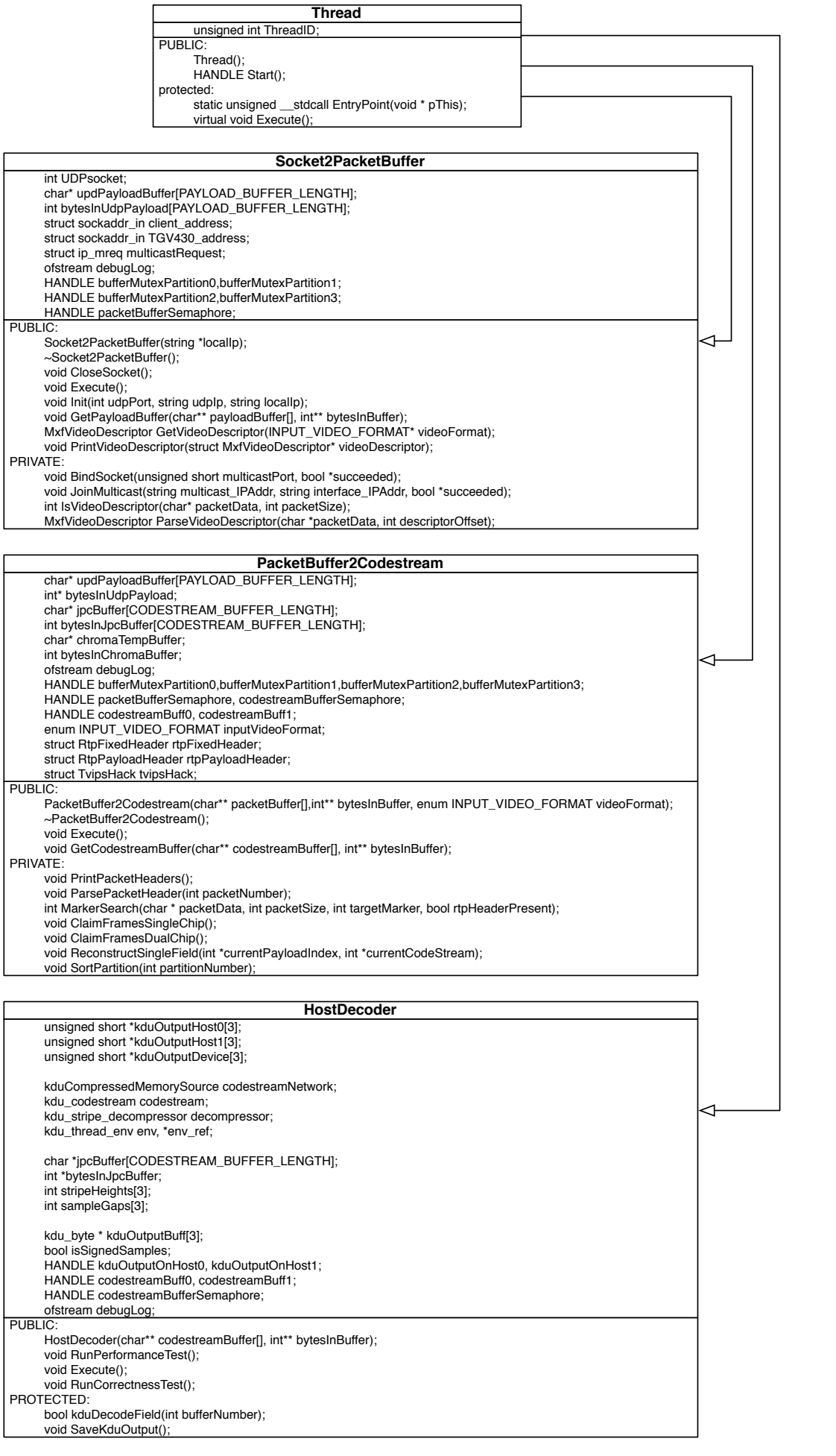
Appendix A

Application Activity Diagram


Thread
main(/gui/deviceDecoder)
socket2PacketBuffer
packetBuffer2Codestream
hostDecoder
start
Socket2PacketBuffer socket2PacketBuffer(..)
packetBuffer2Codestream PacketBuffer2Codestream(..)
Jpeg2000Decoder decoder(..)
HANDLE packetBufferMutex, backEndBufferMutex,
frontEndBufferMutex, fieldTimer
socket2PacketBuffer.start()
packetBuffer2Codestream.start();
Receive UPD packet and put it in the packetBuffer
decoder.start();
Reconstruct compressed frame from multiple UDP
payloads, and put it the codestreamBuffer.
StartGuiMainloop();
Extract compressed frame from codestream buffer.
Do decoding on CPU.
Place result in hostOutputBuffer in host memory.
Other internal threads in gui.
Transfer frame from hostOutputBuffer on host to
deviceInputBuffer on device,.
Do remaining decoding on GPU.
Paint result to display.
end
Window
closed

Appendix B

Application Class Diagram


Thread
unsigned int ThreadID;
PUBLIC:
Thread();
HANDLE Start();
protected:
static unsigned __stdcall EntryPoint(void * pThis);
virtual void Execute();
Socket2PacketBuffer
int UDPsocket;
char* updPayloadBuffer[PAYLOAD_BUFFER_LENGTH];
int bytesInUdpPayload[PAYLOAD_BUFFER_LENGTH];
struct sockaddr_in client_address;
struct sockaddr_in TGV430_address;
struct ip_mreq multicastRequest;
ofstream debugLog;
HANDLE bufferMutexPartition0,bufferMutexPartition1;
HANDLE bufferMutexPartition2,bufferMutexPartition3;
HANDLE packetBufferSemaphore;
PUBLIC:
Socket2PacketBuffer(string *localIp);
~Socket2PacketBuffer();
void CloseSocket();
void Execute();
void Init(int udpPort, string udpIp, string localIp);
void GetPayloadBuffer(char** payloadBuffer[], int** bytesInBuffer);
MxfVideoDescriptor GetVideoDescriptor(INPUT_VIDEO_FORMAT* videoFormat);
void PrintVideoDescriptor(struct MxfVideoDescriptor* videoDescriptor);
PRIVATE:
void BindSocket(unsigned short multicastPort, bool *succeeded);
void JoinMulticast(string multicast_IPAddr, string interface_IPAddr, bool *succeeded);
int IsVideoDescriptor(char* packetData, int packetSize);
MxfVideoDescriptor ParseVideoDescriptor(char *packetData, int descriptorOffset);
PacketBuffer2Codestream
char* updPayloadBuffer[PAYLOAD_BUFFER_LENGTH];
int* bytesInUdpPayload;
char* jpcBuffer[CODESTREAM_BUFFER_LENGTH];
int bytesInJpcBuffer[CODESTREAM_BUFFER_LENGTH];
char* chromaTempBuffer;
int bytesInChromaBuffer;
ofstream debugLog;
HANDLE bufferMutexPartition0,bufferMutexPartition1,bufferMutexPartition2,bufferMutexPartition3;
HANDLE packetBufferSemaphore, codestreamBufferSemaphore;
HANDLE codestreamBuff0, codestreamBuff1;
enum INPUT_VIDEO_FORMAT inputVideoFormat;
struct RtpFixedHeader rtpFixedHeader;
struct RtpPayloadHeader rtpPayloadHeader;
struct TvipsHack tvipsHack;
PUBLIC:
PacketBuffer2Codestream(char** packetBuffer[],int** bytesInBuffer, enum INPUT_VIDEO_FORMAT videoFormat);
~PacketBuffer2Codestream();
void Execute();
void GetCodestreamBuffer(char** codestreamBuffer[], int** bytesInBuffer);
PRIVATE:
void PrintPacketHeaders();
void ParsePacketHeader(int packetNumber);
int MarkerSearch(char * packetData, int packetSize, int targetMarker, bool rtpHeaderPresent);
void ClaimFramesSingleChip();
void ClaimFramesDualChip();
void ReconstructSingleField(int *currentPayloadIndex, int *currentCodeStream);
void SortPartition(int partitionNumber);
HostDecoder
unsigned short *kduOutputHost0[3];
unsigned short *kduOutputHost1[3];
unsigned short *kduOutputDevice[3];
kduCompressedMemorySource codestreamNetwork;
kdu_codestream codestream;
kdu_stripe_decompressor decompressor;
kdu_thread_env env, *env_ref;
char *jpcBuffer[CODESTREAM_BUFFER_LENGTH];
int *bytesInJpcBuffer;
int stripeHeights[3];
int sampleGaps[3];
kdu_byte * kduOutputBuff[3];
bool isSignedSamples;
HANDLE kduOutputOnHost0, kduOutputOnHost1;
HANDLE codestreamBuff0, codestreamBuff1;
HANDLE codestreamBufferSemaphore;
ofstream debugLog;
PUBLIC:
HostDecoder(char** codestreamBuffer[], int** bytesInBuffer);
void RunPerformanceTest();
void Execute();
void RunCorrectnessTest();
PROTECTED:
bool kduDecodeField(int bufferNumber);
void SaveKduOutput();
CudaLib
extern "C" void DecodeFrameOnDevice(int hostBufferNumber);
extern "C" void InitGui(unsigned short *hostOutput0[3], unsigned short *hostOutput1[3], unsigned short *deviceInput[3], int imgWidth, int imgHight);
extern "C" void StartGuiMainloop();
extern "C" void* AllocateMemDevice(unsigned int numBytes);
extern "C" void* AllocateMemHost(unsigned int numBytes);
extern "C" void MemsetDevice(void* buffer, unsigned int numBytes);
extern "C" void FreeMem(void* data);
extern "C" void Cleanup(void);

Appendix C

Irreversible Color Transform

Input to the irreversible color transform considered in this thesis are triplets of color channels from the Y'$C_b C_r$ color space with 8 bit per channel, while the output are triplets residing in the R'G'B' color space. The prime (') symbol means gamma correction is being used. As specified in ITU-R Recommendation BT.601, Y' ranges from 16 to 235, where 16 is the darkest and 235 is the brightest. C_b and C_r range from 16 to 240, where 128 is the zero-point. Y' is often referred to as the luma component, while C_b and C_r often are referred to as the blue-difference and red-difference chroma components.

The first processing to be done, is to scale the luma and chroma components to the correct interval. This is stated in (C.1), together with the resulting intervals.

$$\begin{aligned} y' &= \frac{Y' - 16}{219}, \quad y' \in \Big[0.0, 1.0\Big] \\ c_b &= \frac{C_b - 128}{224}, \quad c_b \in \Big[-0.5, 0.5\Big] \\ c_r &= \frac{C_r - 128}{224}, \quad c_r \in \Big[-0.5, 0.5\Big] \end{aligned} \tag{C.1}$$

The general r'g'b' to y'$c_b c_r$ equations are given in (C.2), (C.3) and (C.4), while the general y'$c_b c_r$ to r'g'b' equation is given in (C.5).

$$y' = K_r \times r' + (1 - K_r - K_b) \times g' + K_b \times b' \tag{C.2}$$

$$c_b = \frac{b' - y'}{2(1 - K_b)} \tag{C.3}$$

$$c_r = \frac{r' - y'}{2(1 - K_r)} \tag{C.4}$$

$$\begin{pmatrix} r' \\ g' \\ b' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 2(1-K_r) \\ 1 & \frac{2(1-K_b)K_b}{1-K_b-K_r} & \frac{2(1-K_r)K_r}{1-K_b-K_r} \\ 1 & 2(1-K_b) & 0 \end{pmatrix} \begin{pmatrix} y' \\ c_b \\ c_r \end{pmatrix} \tag{C.5}$$

$$r', g', b' \in \Big[0.0, 1.0\Big] \tag{C.6}$$

The actual transform coefficients depend on the applied encoding. Since the image content in this setting originally was transmitted over HD-SDI, the transform coefficients are defined in ITU-R Recommendation BT.601 as K_b=0.114 and K_r=0.299. With this, equation (C.5) becomes:

$$\begin{pmatrix} r' \\ g' \\ b' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.402 \\ 1 & -0.34413 & -0.71414 \\ 1 & 1.772 & 0 \end{pmatrix} \times \begin{pmatrix} y' \\ c_b \\ c_r \end{pmatrix} \tag{C.7}$$

The last step in the ICT is to generate full-range R'G'B' triplets, i.e:

$$R' = round(255 \times r') \tag{C.8}$$

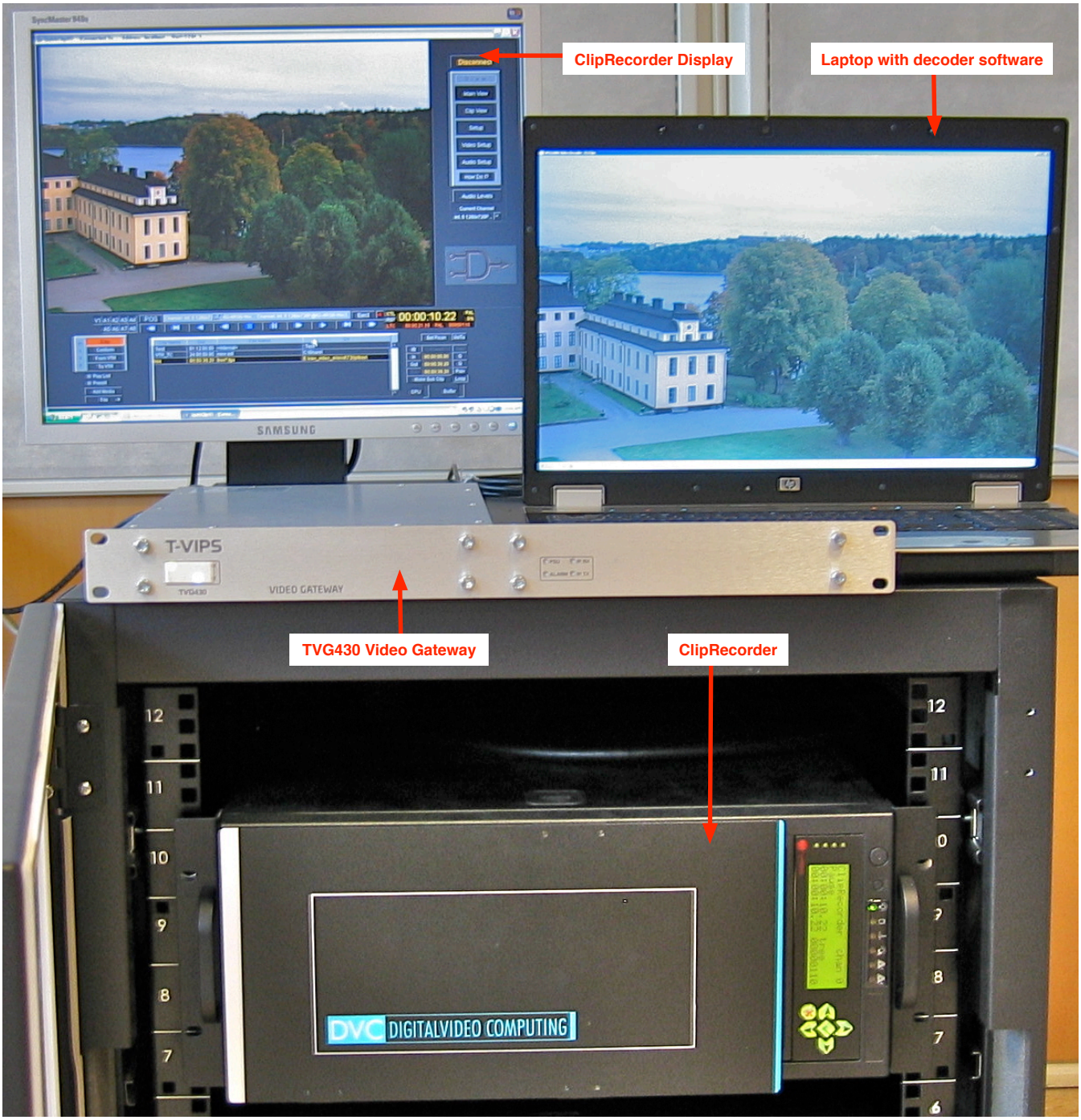
$$G' = round(255 \times g') \tag{C.9}$$

$$B' = round(255 \times b') \tag{C.10}$$

$$R', G', B' \in \Big[0, 255\Big] \tag{C.11}$$

Appendix D

Test Setup


ClipRecorder Display
Laptop with decoder software
TVG430 Video Gateway
ClipRecorder
SAMSUNG
T-VIPS
VIDEO GATEWAY
DVC DIGITALVIDEO COMPUTING

Appendix E

CD-ROM