UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

GHEVC: An Efficient HEVC Decoder for Graphics Processing Units

Diego Felix de Souza

Supervisor: Doctor Leonel Augusto Pires Seabra de Sousa
Co-supervisor: Doctor Nuno Filipe Valentim Roma

Thesis approved in public session to obtain the PhD Degree in Electrical and Computer Engineering

Jury final classification: Pass with Distinction

2018

UNIVERSIDADE DE LISBOA

INSTITUTO SUPERIOR TÉCNICO

GHEVC: An Efficient HEVC Decoder for Graphics Processing Units

Diego Felix de Souza

Supervisor: Doctor Leonel Augusto Pires Seabra de Sousa
Co-supervisor: Doctor Nuno Filipe Valentim Roma

Thesis approved in public session to obtain the PhD Degree in Electrical and Computer Engineering

Jury final classification: Pass with Distinction

Jury

Chairperson: Doctor Isabel Maria Martins Trancoso, Instituto Superior Técnico da Universidade de Lisboa

Members of the Committee:

Doctor Sérgio Manuel Maciel de Faria, Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria

Doctor Gabriel Falcão Paiva Fernandes, Faculdade de Ciências e Tecnologia da Universidade de Coimbra

Doctor Maria Paula dos Santos Queluz Rodrigues, Instituto Superior Técnico da Universidade de Lisboa

Doctor David Manuel Martins de Matos, Instituto Superior Técnico da Universidade de Lisboa

Doctor Nuno Filipe Valentim Roma, Instituto Superior Técnico da Universidade de Lisboa

Funding Institutions

Fundação para a Ciência e a Tecnologia (FCT)
Ph.D. scholarship SFRH/BD/76285/2011

2018

To my beloved daughter Rebecca

Acknowledgments

I wish to express my deep and sincere gratitude for all the support that I have received from my family. Their patience, love, and care were essential to the development of this Thesis. Also, I could not forget all the friends that I have made over these years that contributed to this work with inspiring discussions and fruitful collaboration. A special thanks to Kenan Turbić, Diogo Mera, Nuno Neves, José Germano, Lídia Kuan, Svetislav Momcilović, Jelena Milošević, Frederico Pratas, Mauricio Ramalho, Vladimir Ivannikov, Pham Tien Cuong, Sanchit Singh, Janine Iser, Dhiraj Shah, Berner Panti, Diego Batista, Daniel Albuquerque, Bruno Almeida, Bernardo Pereira, Antônio Neto, Felipe Soares, Otto Soares, Rodrigo Ferreira, Francisco Seixas, Luís Lucas, Danillo Graziosi, João Silva and Nerminko Omanić. I would like to name you all, but unfortunately, I have insufficient space for all of you.

I also want to mention namely our Signal Processing Systems (SiPS) group, in particular, Professor Pedro Tomás and Ana Jesus. I would like to thank the Instituto de Engenharia de Sistemas e Computadores - Investigação e Desenvolvimento (INESC-ID) Lisboa and all the Departamento de Engenharia Electrotécnica e de Computadores from the Instituto Superior Técnico. My gratitude goes also to Spin Digital Video Technologies team for all the support.

I would like to acknowledge the Fundação para a Ciência e a Tecnologia (FCT) for the financial assistance with the Ph.D. scholarship SFRH/BD/76285/2011.

Special thanks go to Professor Aleksandar Ilić, his contribution was crucial for my work and personal growth, whom I have deeply displeased due to my behavior. I hope one day he can forgive all my mistakes.

Finally, I would also like to thank Professor Leonel Sousa and Professor Nuno Roma for welcoming me into their research group in one of my worst moments in life and for their guidance.

Abstract

The compression efficiency achieved with the High Efficiency Video Coding (HEVC) standard comes at the cost of a significant increase of the computational load at both the encoder and the decoder. When considering HEVC decoders, such an increased burden is a limiting factor to accomplish real-time, especially for high definition video sequences (e.g. Ultra HD 4K and beyond). On the other hand, modern Graphics Processor Units (GPUs) have evolved into programmable and powerful parallel accelerators, being able to deliver an execution performance that greatly exceeds the capabilities of multi-core Central Processing Units (CPUs). However, this performance is mostly attainable for compute-intensive applications, with significant amount of data parallelism and regular memory access patterns. Accordingly, fully exploiting the GPU capabilities for a set of diverse and computationally complex HEVC decoding procedures is far from being a trivial task. In this scenario, the presented research is focused on developing an efficient GPU-based HEVC decoder, denoted as GHEVC. Such a data-parallel GHEVC decoder integrates the whole decompression pipeline, where the several HEVC procedures are executed in a highly heterogeneous environment composed by a CPU and a GPU. The herein presented algorithms comprehensively exploit both coarse and fine-grained parallelization opportunities in an integrated perspective by re-designing the execution pattern of the involved HEVC procedures, while simultaneously coping with their inherent computational complexity and data dependencies. As a result, the proposed GHEVC decoder, which is fully compliant with the HEVC standard, has showed to be a remarkable approach, being capable of satisfying hard real-time requirements to process Ultra HD 4K video sequences.

Keywords

Parallel Processing, Graphics Processor Units (GPUs), High Efficiency Video Coding (HEVC), Video Decoding, Real-time.

Resumo

A eficiência de compressão alcançada pela norma High Efficiency Video Coding (HEVC) é conseguida à custa de um aumento significativo da carga computacional no codificador e no descodificador. Para descodificadores HEVC, o aumento da carga computacional é um fator limitativo para a execução em tempo real, especialmente para sequências de vídeo de alta definição (Ultra HD 4K). Por outro lado, as Unidades de Processamento Gráfico (GPUs) evoluíram significativamente ao longo dos últimos anos, sendo hoje poderosos aceleradores programáveis, capazes de atingir um nível de desempenho que excede em muito o de uma Unidade de Processamento Central (CPU) multinúcleo. No entanto, este nível de desempenho é, em geral, atingido para aplicações de computação intensiva, com um alto grau de paralelismo e acessos regulares à memória. Assim, a exploração efetiva das capacidades de processamento das GPUs para os procedimentos da norma HEVC, que são computacionalmente complexos, está longe de ser uma tarefa fácil. Neste cenário, o trabalho apresentado foca-se na investigação e desenvolvimento de um descodificador HEVC computacionalmente eficiente, baseado na utilização de GPUs, a que se designou de GHEVC. O descodificador GHEVC inclui todos os procedimentos do HEVC, sendo estes executados num sistema heterogéneo constituído por um CPU e uma GPU. Desta forma, os algoritmos aqui apresentados exploram oportunidades de paralelismo numa perspectiva integrada, através de uma nova execução dos procedimentos do descodificador HEVC, os quais lidam com as suas inerentes complexidades e dependências computacionais. Como resultado, o descodificador GHEVC, totalmente compatível com a norma HEVC, é capaz de satisfazer os rigorosos requisitos de tempo real para processar sequências de vídeo Ultra HD 4K.

Palavras Chave

Processamento Paralelo, Unidades de Processamento Gráfico (GPUs), High Efficiency Video Coding (HEVC), Descodificação de Vídeo, Tempo Real.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
BB	Boundary Block
BF	Bypass Flag
BS	Boundary filtering Strength
CABAC	Context-Adaptive Binary Arithmetic Coding
CBF	Coded Block Flag
CPU	Central Processing Unit
CTU	Coding Tree Unit
CU	Coding Unit
CUDA	Compute Unified Device Architecture
DBF	Deblocking Filter
DCT	Discrete Cosine Transform
DIT	De-quantization and Inverse Transform
DRAM	Dynamic Random-Access Memory
DSP	Digital Signal Processor
DST	Discrete Sine Transform
DVFS	Dynamic Voltage Frequency Scaling
FPGA	Field-Programmable Gate Array
FPS	frames per second
GHEVC	GPU-based HEVC
GPU	Graphics Processing Unit

HD	High Definition
HEVC	High Efficiency Video Coding
HIP	Heterogeneous-compute Interface for Portability
HM	HEVC Test Model
ILP	Instruction-level parallelism
IP	Intra Prediction
ISO/IEC	International Standardization Organization/International Electrotechnical Commission
ITU	International Telecommunications Union
JCT-VC	Joint Collaborative Team on Video Coding
Mbps	megabit per second
MC	Motion Compensation
MPEG	Moving Picture Experts Group
OpenCL	Open Computing Language
OpenGL	Open Graphics Library
OpenMP	Open Multi-Processing
OWF	Overlapped Wavefront
PB	Prediction Block
PU	Prediction Unit
QP	Quantization Parameter
RVC-CAL	Reconfigurable Video Coding CAL Actor Language
SAO	Sample Adaptive Offset
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction Multiple Thread
SM	Streaming Multiprocessor
SoC	System on Chip
SVT	Sveriges Television AB

ThB Thread Block

TB Transform Block

TBF Transquant Bypass Flag

TSF Transform Skip Flag

TU Transform Unit

VCEG Video Coding Experts Group

VoD Video on Demand

WPP Wavefront Parallel Processing

1

Introduction

Contents

In 2015, digital video transfers were reported to be responsible for around 70% of the total global internet traffic [1]. In general, those video sequences correspond to: Internet video (e.g., YouTube[1] and Hulu[2]); live Internet video; Internet video to TV (e.g., Netflix[3]); online video purchases and rentals; video conference; and web-based video monitoring. Although the supply of good quality video contents is still low, the demand for High Definition (HD) videos is extremely large, which includes video resolutions such as Full HD (1920×1080 pixels) and beyond (Ultra HD). In fact, Ultra HD video sequences are expected to be 20.7% of all Video on Demand (VoD) traffic in 2020 [1], when considering Ultra HD 4K (3840×2160 pixels) and Ultra HD 8K (7680×4320 pixels) resolutions. This demand for higher video resolutions in applications and services has led to a consequent growth for bandwidth and storage requirements, which results in greater demands for higher video compression rates and more advanced video coding mechanisms.

In order to satisfy these needs, the Joint Collaborative Team on Video Coding (JCT-VC) [2] was created in 2010 to develop a new standard. Such consortium integrated several video coding experts from: *i*) International Telecommunications Union (ITU)-Telecommunication Standardization Sector Study Group 16 Video Coding Experts Group (VCEG); and *ii*) International Standardization Organization/International Electrotechnical Commission (ISO/IEC) JTC 1/SC 29/WG 11 Moving Picture Experts Group (MPEG). The requirements of this new video standard were defined and the corresponding joint call for proposals was published in the same year [3]. In 2013, the first version of the new video standard [4], designated as High Efficiency Video Coding (HEVC), was released, particularly focused on two key issues: increase the video resolution and allow parallel processing [5]. By relying on cutting-edge encoding techniques, the newest video coding standard has shown to reduce approximately by half the bitrate required to compress a video sequence with the same visual quality [6, 7], when compared to its predecessor H.264/MPEG-4 AVC standard [8] with High profile [9], which was one of the most applied video coding standards in 2013.

As a result, the HEVC standard does not only provide a more efficient utilization of the storage resources, but it also offers a higher suitability to Ultra HD 4K and Ultra HD 8K video resolutions, which are seen as near-future targets for video services. Despite its higher compression efficiency in comparison with the H.264/MPEG-4 AVC standard, the computational complexity of optimized HEVC and H.264/MPEG-4 AVC decoders are not significantly different [10, 11] (no more than 2× slower). Nevertheless, the decoding procedures of Ultra HD video sequences greatly increase the computational load of HEVC decoders, mainly due to the larger amount of data to be processed when compared with more modest video resolutions. In this way, to achieve real-time capabilities for Ultra HD video sequences in HEVC decoders, it is highly required to exploit the available parallelism either in hardware or in software [12]. In particular, HEVC decoders implementations can take advantage of multi- or many-core architectures by employing all the parallelization models

[1] [Online] Available: https://www.youtube.com
[2] [Online] Available: http://www.hulu.com
[3] [Online] Available: https://www.netflix.com

that are made available by the HEVC standard [13, 14], which are categorized in:

- **Sequence-level parallelization**: processes several pictures at the same time, provided that their temporal dependencies are satisfied.

- **Picture-level parallelization**: simultaneously decodes independent parts of the picture, where the number and size of these independent parts are set on the encoder side.

- **Block-level parallelization**: decompresses non-overlapping procedures or parts of a pixel block, where a pipeline scheme is elaborated for different decoding procedures; or non-overlapping parts of the pixel block for the same decoding procedure are processed in parallel.

Each of these models provides different levels of parallelism, which can be exploited to increase the performance of hardware HEVC decoders [15], as well as software-based decoders for multi- or many-core Central Processing Units (CPUs) [16]. Nonetheless, each chosen architecture and parallelization model has its own advantages and limitations.

When considering the limitations of the encoder side, a set of HEVC features and parameters (e.g., motion vectors, block partitioning and quantization step) can be carefully selected and configured to achieve the best trade off between distortion, compression rate and computational complexity [17]. However, high compression rates are only attained at the cost of a corresponding high computational load, since the encoded bitstream is usually generated by taking advantage of the main HEVC compression features, such as: *i*) several partitioning modes, including asymmetric partitioning to adapt to the video content; *ii*) 35 intra prediction modes allied with quarter-pel motion vectors and interpolation filters, to exploit spatial and temporal correlation; *iii*) different transform types and block sizes (from 32×32 to 4×4) to reduce the residual data redundancy; and *iv*) in-loop filtering, to remove block artifacts and sample distortion. Furthermore, although the encoder may select the coding tools which it considers to be most suitable, an HEVC decoder has to be able to decompress any compliant bitstream, for the defined set of profiles, levels and tiers [18], regardless of the involved computational complexity.

Hence, the specified profile, level and tier of an HEVC decoder shall indicate the available processing capabilities to decode a bitstream. In particular, a set of coding tools that may be used to encode a video sequence into a bitstream is defined by the profile. The level and tier, among other settings, limit the maximum picture size, the picture buffer sizes, and the bit rates. Therefore, designing a decoder with the ability to comply not only with the computationally complex HEVC operations, but also with the strict frame rate requirements, is far from being a trivial task.

1.1 Motivation

Hardware-based HEVC decoder implementations usually target specific profiles, levels and tiers, in order to achieve low power consumption. Nevertheless, their decoding capabilities are often limited to a few frame resolutions and rates. Moreover, hardware-based HEVC decoders are not present in all computing devices, where they are mostly found in System on Chips (SoCs) for embedded and mobile devices (e.g., Qualcomm Snapdragon 820[4]). When considering laptop and desktop devices, hardware-based HEVC decoders are also included in the latest processors (e.g., Intel Skylake[5]). A remarkable drawback of hardware accelerated decoders is to take up valuable die space on a SoC. Furthermore, hardware-based decoders can't be redesigned after production, while reconfigurable hardware, e.g., Field-Programmable Gate Array (FPGA), despite being more flexible, lead to higher power consumption when compared to dedicated hardware acceleration.

On the other hand, software-based HEVC decoders are not limited by specific profiles, levels and tiers. However, software approaches are usually limited by the architecture capabilities and the amount of parallelism exposed by the algorithm. Usually, software-based HEVC decoders take only advantage of the CPU and they often exploit Single Instruction, Multiple Data (SIMD) instruction sets [16], as well as multithreading [19]. By only exploiting the CPU, most software decoders do not employ the full capabilities of current computing devices, which are often composed by heterogeneous systems, consisting of a CPU and a Graphics Processing Unit (GPU).

In the past decade, GPUs have evolved from a fixed-function graphics pipeline to a programmable and general purpose parallel processor, with the offered computing power often exceeding the processing capabilities of multi-core CPUs [20]. Nevertheless, while the CPU is predominantly designed and optimized for sequential code performance, the GPU is specialized for compute-intensive and highly parallel data-level computation [21], like 3D rendering. Hence, the GPU has been designed for intense data processing, rather than data caching and control flow. In what concerns the processing acceleration, modern GPUs can achieve performance levels that greatly exceed the capabilities of multi-core CPUs [21].

1.2 Main Objectives and Contributions

Although GPUs have evolved to programmable and powerful parallel accelerators, they are mostly suitable for compute-intensive applications with a high parallelism degree. In this way, to fully exploit the GPU architecture, the targeted application has to be conveniently redesigned, in order to maximize the degree of parallelism and to take advantage of the GPU memory hierarchy and high execution concurrency. As a result, fully exploiting the GPU capabilities for a set of diverse and computationally complex HEVC decoding procedures is usually a difficult task.

[4] [Online] Available: https://www.qualcomm.com/products/snapdragon/processors/820
[5] [Online] Available: https://software.intel.com/en-us/blogs/2015/12/11/codecs-are-they-slowing-you-down

Hence, the main objective of this Thesis is the investigation of a comprehensive GPU-based HEVC (GHEVC) decoder, which will ensure a fully compliant and real-time HEVC decoding for Ultra HD video sequences, by offloading most of the HEVC procedures to the GPU. The main contributions of this Thesis are summarized as follows.

- A comprehensive redesign of all HEVC decoder procedures, in order to decode a video sequence in GPU accelerators, which implies a fully exploitation of the available parallelism, optimization of the memory access and increase of the instruction throughput.
- A unified design of all the GHEVC components, by reinforcing data sharing among different HEVC procedures by taking advantage of the GPU's memory hierarchy.
- A frame-level GPU parallel processing scheme, where different parts of the frame are processed in parallel, while ensuring the HEVC standard compliance.

With these goals in mind, it can be identified the set of contributions presented in this Thesis, which already led to the following publications in international journals and conferences:

- **Journals**
  - [22]: D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GHEVC: An efficient HEVC decoder for graphics processing units. *IEEE Transactions on Multimedia*, 19(3):459–474, Mar. 2017. ISSN 1520-9210. doi:10.1109/TMM.2016.2625261.
  - [23]: B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU parallelization of HEVC in-loop filters. *International Journal of Parallel Programming*, pages 1–21, 2017. ISSN 1573-7640. doi:10.1007/s10766-017-0488-z.
  - [24]: D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU-assisted HEVC intra decoder. *Journal of Real-Time Image Processing*, 12(2):531–547, 2016. ISSN 1861-8219. doi:10.1007/s11554-015-0519-1.
- **Conferences**
  - [25]: B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. Efficient HEVC decoder for heterogeneous CPU with GPU systems. In *2016 IEEE 18th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6, Sept. 2016. doi:10.1109/MMSP.2016.7813353.
  - [26]: D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU acceleration of the HEVC decoder inter prediction module. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1245–1249, Dec. 2015. doi:10.1109/GlobalSIP.2015.7418397.

  - [27]: D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. HEVC in-loop filters GPU parallelization in embedded systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 123–130, July 2015. doi:10.1109/SAMOS.2015.7363667.
  - [28]: D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. Towards GPU HEVC intra decoding: Seizing fine-grain parallelism. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, June 2015. doi:10.1109/ICME.2015.7177515.
  - [29]: D. F. de Souza, N. Roma, and L. Sousa. OpenCL parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms. In *22nd European Signal Processing Conference (EUSIPCO)*, pages 755–759, Sept. 2014.
  - [30]: D. F. de Souza, N. Roma, and L. Sousa. Cooperative CPU+GPU deblocking filter parallelization for high performance HEVC video codecs. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4993–4997, May 2014. doi:10.1109/ICASSP.2014.6854552.

- **Accepted ()under production)**

  - [31]: B. Wang, D. F. de Souza, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, A. Ilic, N. Roma, and L. Sousa. Highly parallel HEVC decoding for heterogeneous systems with CPU and GPU. *Signal Processing: Image Communication*, 2018.

1.3 Outline

This dissertation is organized in five chapters, according to the following outline.

- **Chapter 2**: introduces the basic principles of the GPU architecture and programming, as well as a brief overview of the memory hierarchy and optimization techniques.
- **Chapter 3**: provides an overview of the functional principles behind the HEVC standard, as well as a review of the state-of-the-art parallel HEVC procedures and decoder implementations.
- **Chapter 4**: presents a detailed description of the proposed approach for the parallelization of the HEVC decoder, which was denoted GHEVC. GHEVC modules integration are also comprehensively presented, which includes the chosen frame-level parallel processing.
- **Chapter 5**: discusses the profiling and evaluation of the implemented GHEVC decoder, where the obtained experimental results are presented and discussed for each individual module.
- **Chapter 6**: draws the main conclusions of this Thesis and the final remarks for future work.

2

Overview of Graphics Processing Units

Contents

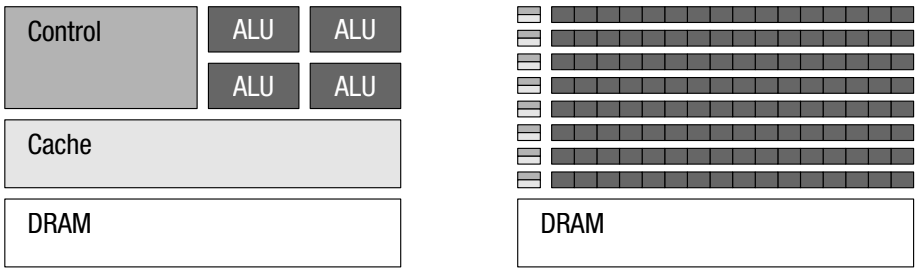
Along the past years, multi-core heterogeneous systems have proven to be able to offer more performance or energy efficiency than the homogeneous multi-core counterparts. When considering heterogeneous systems consisting of CPU and GPU devices, it has been shown that further computational gains can be achieved when the best features of both devices are exploited cooperatively [32]. Since the goal of this research is to accelerate all possible HEVC procedures by exploiting GPU devices, the GPU main characteristics and capabilities are comprehensively discussed in this chapter.

In 1999, the NVIDIA GeForce 256 GPU [33] was made available in the market. Contrasting to the former graphics cards, the NVIDIA GeForce 256 GPU was the first to include a single-chip 3D real-time graphics processor with a configurable 32-bit floating-point vertex transform and lighting processor, which were programmed through Application Programming Interfaces (APIs), such as Open Graphics Library (OpenGL) and Microsoft DirectX 7 (DX7).

As programmable GPU cores evolved, GPUs became more flexible and easy to program. Nevertheless, those GPUs were mainly developed for graphics processing purposes, meaning that they were very difficult to use because programmers had to use the equivalent of graphics APIs, such as OpenGL, to access the GPU cores. For example, the work in [34] addresses the overflow and rounding problems in video decoding interpolation, implemented by a pipeline of vertex/pixel shader procedures. In [35], a stream-based computing model was proposed in order to incorporate GPUs into GRID environment and use them to replace CPUs as the main computing devices.

In November 2006, NVIDIA released the Compute Unified Device Architecture (CUDA) [36], which included several new features to facilitate general-purpose computing on GPUs. On the other hand, in 2009, the Open Computing Language (OpenCL) [37] framework arose as a highly viable alternative to exploit task-parallelism and data-parallelism across multiple heterogeneous devices, including both CPUs and GPUs. Although OpenCL offers code portability among different GPU manufacturers, only with CUDA it is possible to fully exploit the capabilities of current NVIDIA GPU architectures and to ensure a fine-grain control over program executions. In particular, although the tested NVIDIA devices (presented in Chapter 5) fully support the OpenCL version 1.2 API [38], some of the CUDA functions and configurations are not supported in this OpenCL specification. Nevertheless, although CUDA terminology shall be extensively adopted in this dissertation, the provided explanations regarding the main functional principles and parallelization strategies for the proposed GHEVC decoder are oblivious to the programming model. In fact, since the OpenCL specification largely relies on the NVIDIA CUDA legacy, the proposed approaches can be easily ported to the OpenCL API with only slight modifications.

Regardless the considered framework (CUDA or OpenCL), it is important to notice that while the CPU is conceived and optimized to exploit Instruction-level parallelism (ILP) or sequential application performance, the GPU device is designed for applications with a specific set of characteristics [39], such as: high computational requirements, high level of data parallelism and



(a) CPU. **(b)** GPU.

Figure 2.1: High level representation of CPU and GPU architectures.

prioritizing throughput over latency. In this way, the GPU architecture devotes more transistors to data processing rather than data caching and control flow [36], as it is represented in Figure 2.1, which depicts a simplified representation of both architectures. Hence, the same program (denoted as *kernel* in GPU terminology) is executed on many data elements in parallel, where there is a lower requirement for advanced flow control. Also, since the computational requirements are large, the memory access latency can be hidden with calculations instead of relying on large caches. Hence, the higher the level of data parallelism and computational load, the better the GPU performance over the CPU's.

2.1 Programming Model

The GPU programming model is designed to run the same program (kernel) across hundreds or thousands of concurrent threads and cores [40]. Hence, a CUDA application is organized into a *host*[1] program, consisting of one or more sequential threads running on the *host* CPU, and one or more parallel kernels that are conveniently described for execution on the parallel processing *device* (GPU). To achieve this parallel processing capability, each kernel defines a multidimensional *grid* of *thread blocks* according to its application/algorithm requirements. The threads in a Thread Block (ThB) are also organized in a multidimensional way, where each thread as a unique index. This provides a natural way to invoke computation across the elements in a regular domain, such as a vector or a matrix [36].

The threads in a ThB are assumed to be executed in the same processor and to share memory resources. However, since the processor resources are limited, there is a limit for the number of threads in a ThB. In current GPUs, a ThB may contain up to 1024 threads. However, a kernel can be executed by multiple equally-shaped ThB, so that the total number of threads is equal to the number of threads per ThB times the number of ThBs in a grid [36].

Although the threads inside a ThB can interact and cooperate among themselves through

[1] For the sake of clarity, in the remaining of this Thesis the CPU is denoted as the *host*, whereas the GPU is denoted as the *device*.

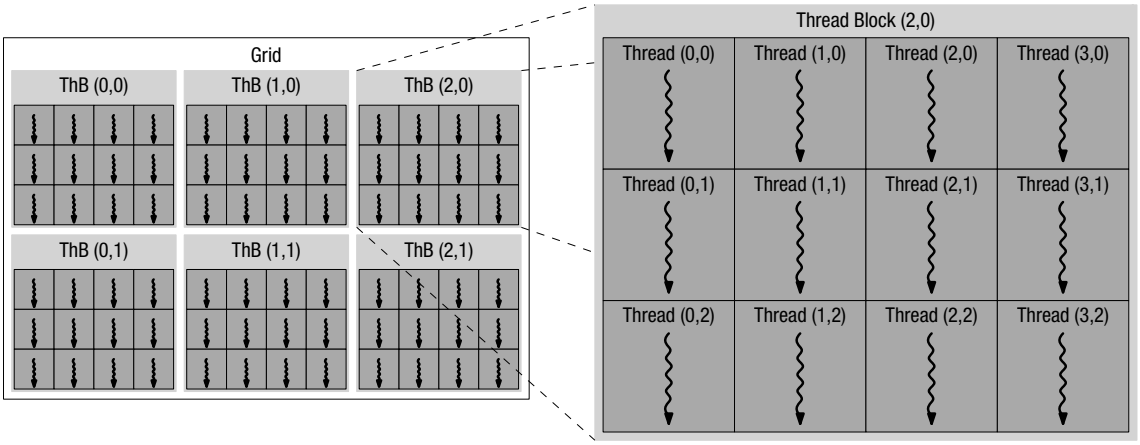


Figure 2.2: An example of a grid of ThBs.

barrier synchronization and shared access to a ThB private memory space, the ThBs are required to be executed independently of the other ThBs [41]. This means that it must be possible to execute them in any order, in parallel or in series. This independence requirement allows ThBs to be scheduled in any order across any number of cores, making the programming model scalable across an arbitrary number of cores, as well as across a variety of parallel architectures. For example, Figure 2.2 presents a kernel grid with 6 ThBs, where each ThB has 12 threads. If a GPU processor can handle only one ThB at a time, the same kernel would run up to 6$\times$ faster in a GPU with 6 processors than on a GPU with just one, without any kernel modification by the programmer.

2.1.1 Host and Device Synchronization

According to the CUDA programming model, the kernel (computation) and memory transfers (host to device and device to host) are organized in *streams* [36]. A stream is a sequence of commands (kernels and memory transfers) that are executed in order. On the other hand, streams may execute concurrently or out of order with respect to each other (according to the device resources and capabilities). In this way, concurrent operations, such as overlapped kernels and memory transfers are achieved with multiple streams.

The kernels are always issued asynchronously, which means that the host thread does not wait until a kernel finishes its execution. On the other hand, the memory transfers between host and device may be synchronous or asynchronous.

The programming model also defines explicit and implicit barrier synchronizations between the host and the device. Explicit synchronizations include barriers (among others): *i*) at device-level (*cudaDeviceSynchronize*), for which the host thread waits until all preceding commands in all streams of all host threads have completed; and *ii*) at stream-level (*cudaStreamSynchronize*), for which the host thread waits until all preceding commands in the given stream have completed, while allowing other streams to continue executing on the device.

2.1.2 Threads Synchronization

The threads of a single thread block are allowed to synchronize with each other via barrier synchronizations (e.g. *__syncthreads*). Hence, as soon as the device thread reaches one of these barriers, it waits until all threads in the ThB have reached that point and all memory accesses have been made by these threads prior to the barrier is visible to all threads in the ThB. Thus, inter-ThB barrier synchronization is used to coordinate communication between the threads of the same ThB. For example, when some device threads within a ThB access the same addresses in the device's memory, there are potential read-after-write, write-after-read, or write-after-write hazards. These data hazards can be avoided by synchronizing threads in-between these accesses.

2.1.3 Memory Fence Functions

The programming model described here assumes a device with a weakly-ordered memory model, which means: *i*) the order in which a device thread writes data to the device or host memory is not necessarily the order in which the data is observed as being written by another device or host thread; and *ii*) the order in which a device thread reads data from the device or host memory is not necessarily the order in which the read instructions appear in the program for instructions that are independent to each other. To solve this problem, memory fence functions can be used to enforce some ordering, when threads consume some data produced by other threads in the same kernel.

In the CUDA programming model, three memory fence functions are available [36] (according to the device capabilities):

1. *__threadfence_block*: ensures that all writes (or reads) to the memory made by the calling thread before the call to *__threadfence_block* are observed by all threads in the same ThB as occurring before all writes (or reads) to the memory made by the calling thread after the call to *__threadfence_block*.

2. *__threadfence*: acts as *__threadfence_block* for all threads in the ThB of the calling thread and also ensures that no writes to the device memory made by the calling thread after the call to *__threadfence* are observed by any thread in the device as occurring before any write to the device memory made by the calling thread before the call to *__threadfence*.

3. *__threadfence_system*: acts as *__threadfence_block* for all threads in the ThB of the calling thread and also ensures that all writes (or reads) to the device or host memory made by the calling thread before the call to *__threadfence_system* are observed by all threads in the device, host threads, and all threads in peer devices as occurring before all writes (or reads) to the device or host memory made by the calling thread after the call to *__threadfence_system*.

It is important to notice that memory fence functions only affect the order of memory operations

on a thread. Hence, they do not ensure that these memory operations are visible to other threads. The memory operations are visible only if the other threads truly observe the device memory and not cached versions of it [36], which can be ensured by using the *volatile* keyword.

2.2 Memory Hierarchy

In order to provide parallel processing and scalability capabilities, CUDA threads may access data from multiple memory spaces, where each one has its own characteristics, such as size, latency and bandwidth. Those memory spaces are exposed to the programmer to store data in the most performance-optimal way. Therefore, the data accesses must be carefully managed, in order to efficiently use the complex GPU memory hierarchy and to achieve high performance.

2.2.1 Host Memory

Although the host memory does not belong to the GPU, the host memory can be accessed by kernels: it is called *zero-copy* memory space. Zero-copy is a feature that was introduced in version 2.2 of the CUDA Toolkit. A kernel is not allowed to allocate or free zero-copy memory, but may use pointers to such zero-copy memory space passed in from the host program. By doing so, the GPU threads can directly access the host memory as long as it is mapped/pinned to a non-pageable memory. On integrated GPUs, where the host and the device memory are physically the same, mapped/pinned memory provides always gains in performance because it avoids superfluous copies. On discrete GPUs, mapped/pinned memory is advantageous only in certain cases, since data is not cached on the GPU. In this case, performance gain may be obtained if the data is going to be accessed only once and if the data accesses are coalesced.

2.2.2 Global Memory

The global memory is an off-chip large memory space that is visible to all threads. Since the global memory is usually implemented by an external Dynamic Random-Access Memory (DRAM), its latency can be hundreds of processor clocks. The host holds the responsibility for managing the global memory (e.g. to allocate memory space), as well as for data transfers between the host memory and the device memory space. In this global memory, it is possible to share data between threads of different kernels.

Although the order of memory reads and writes to the same address is preserved within a thread, the order of accesses to different addresses may not be preserved. In fact, accesses to the same global memory address by different threads are not guaranteed to have sequential consistency, since the ThB execution order is unknown. Nevertheless, *Memory Fence Functions* can be used to coordinate the global memory accesses. Moreover, atomic functions are available to guarantee memory accesses without interference from other threads.

2.2.3 Constant Memory

The constant memory is a read-only memory space that is also accessible by all threads. Only the host can manage and store values in the constant memory space. Hence, just like the global memory, the constant memory is persistent across kernel launches by the same application. From the device perspective, the constant memory is immutable and may not be modified. In this way, the host must set the constant memory variables prior to launching a kernel.

2.2.4 Texture Memory

The texture memory is a read-only memory space larger than the constant memory and it is also visible by all threads. This memory is meant to store immutable arrays. As the constant memory, only the host can manage and store values in the texture memory. However, contrasting to the global and constant memory spaces, texture memory offers different addressing modes, as well as data filtering for some specific data formats [36]. Texture fetches are cached in the processor cache hierarchy designed to optimize the throughput of texture fetches from thousands of concurrent threads. Since the texture memory (and the associated cache) was designed to be used with 3D graphics, by conveniently storing 2D arrays, they can be efficiently used as a better cache of the global memory for specific non-aligned memory accesses.

2.2.5 Local Memory

The local memory is a private memory space that is visible only to a single thread. The local memory space resides in the device memory (off-chip), so local memory accesses have the same high latency and low bandwidth as global memory accesses. The local memory is architecturally larger than the thread's register file. Nevertheless, the compiler is likely to place several variables in the local memory, such as: *i*) arrays for which it cannot determine that they are indexed with constant values; *ii*) large structures or arrays that would consume too much register space; and *iii*) any variable, whenever the kernel uses more registers than available (this is also known as *register spilling*) [36].

2.2.6 Shared Memory

The shared memory is an on-chip memory (much like an L1 cache) with low access latency and high bandwidth, which is visible to all threads in the same ThB. In this way, the shared memory traffic does not need to compete with limited off-chip bandwidth (e.g. for accessing global memory). Hence, it is practical to accommodate very high bandwidth memory structures on-chip to support the read/write demands. Moreover, variables and data structures stored in the shared memory exist from the time a ThB is created to the time it terminates.

The memory accesses in this memory space are also not ordered. Therefore, just like the global memory, atomic functions are also permitted in the shared memory. Moreover, synchronization points and memory fences can also guarantee the correct memory access order and coordinate the accesses between threads in the same ThB.

2.3 Computational Architecture

For the sake of simplicity, only the NVIDIA GPU architecture details are going to be presented. Similar characteristics may be found on GPUs from other vendors. The NVIDIA GPU architectures are mainly composed of a scalable array of multithreaded Streaming Multiprocessors (SMs) [36]. When a kernel is issued by one stream, the ThBs of the grid are enumerated and distributed to the available SMs and with execution capacity. According to the SM capability, multiple ThBs can be concurrently executed on one SM and all the threads in a ThB are executed in only one SM. As soon as the ThBs finish their execution, new blocks are launched on the vacated SMs.

Since the SMs are designed to be highly multithreaded [42], it is possible to: *i*) hide the latency of memory accesses; *ii*) support fine-grained parallel graphics shader and computing programming models; *iii*) virtualize the physical processors as threads and ThBs to provide transparent scalability; and *iv*) simplify the parallel programming model, by preparing a serial program (kernel) for one thread. Hence, to manage and efficiently execute such a large amount of threads, the SM employs an architecture model designed as Single Instruction Multiple Thread (SIMT).

With SIMT, the SM is responsible for creating, managing, scheduling, and executing threads in groups of 32 parallel threads called *warps*[2]. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and therefore can branch and execute independently [36]. Whenever a ThB is assigned to be run in a SM, the threads are split into warps and each warp gets scheduled by a warp scheduler of the SM for execution. Since a warp executes one common instruction at a time, the number of threads in a ThB should be multiple of 32, in order to avoid inactive device threads.

Moreover, full efficiency can only be achieved when all 32 threads of a warp agree on a common execution path. As soon as one thread in a warp diverges due to a conditional branch, the warp executes serially each execution path, by disabling all the threads that do not belong to that path. When all execution paths are completed, the threads shall converge back to the same execution path. Nevertheless, since different warps execute independently regardless of whether they are following common or disjoint execution paths, branch divergence degrades the performance only when it occurs with threads within a warp.

In terms of performance optimization, three main strategies should be taken into account in the

[2] In the OpenCL framework, the warp concept is defined as *wavefront*. In AMD GPUs, for example, some of the low-end and older GPUs, such as the AMD Radeon HD 54XX series graphics cards, have a *wavefront* size of 32 threads. Higher-end and newer AMD GPUs have a *wavefront* size of 64 parallel threads. [43]

development of the kernels, according to the SM architecture:

1. **Maximize the degree of parallelism**: each kernel should be implemented in such a way that it exposes as much data parallelism as possible, allowing a large number of simultaneously active threads.

2. **Instruction Throughput**: divergence control flow instructions in the threads of a warp (warp divergence) should be avoided in order to achieve maximum instruction throughput.

3. **Memory Optimizations**: data accesses of each module should be carefully managed, in order to efficiently take advantage of the complex GPU memory hierarchy, i.e., global, cache, shared, register, constant and texture memories. Moreover, memory access latency, coalesced accesses, bank conflicts, register spilling and memory bandwidth utilization should also be taken into account.

In fact, it should be taken into account that the amount of registers and shared memory available in the SM is limited. Hence, depending on the amount of registers and shared memory requested by the kernel, the number of ThBs and warps which can reside in a SM should be defined. If there are not enough registers or shared memory available per SM to process at least one ThB, the kernel will not be executed. These limits, as well the amount of registers and shared memory available on the SM, depend on the capability of the device.

Modern NVIDIA GPU architectures are also defined by their *compute capability*. The compute capability of a device is represented by a version number, sometimes also called its "SM version". This version number identifies the set of features supported by the GPU hardware and it is used by applications (at runtime) to determine which hardware features and/or instructions are available on the present GPU.

The compute capability comprises a major revision number X and a minor revision number Y and it is denoted by $X.Y$. Devices with the same major revision number have the same core architecture. The major revision number is 5 for devices based on the *Maxwell* architecture, 3 for devices based on the *Kepler* architecture, 2 for devices based on the *Fermi* architecture, and 1 for devices based on the *Tesla* architecture. The minor revision number corresponds to incremental improvements to the core architecture, possibly including new features.

Table 2.1 presents several technical details of NVIDIA GPUs. For example, *warp shuffle functions* are only available for compute capability 3.0 or higher. These functions, which were introduced by NVIDIA Kepler architecture, allow to directly share data between threads of the same warp without employing the shared memory. Hence, threads of a warp can read each others' registers by using a new instruction called *shuffle*. This functionality is not currently available in the OpenCL framework.

The *Throughput of 32-bit floating-point operations* presented in Table 2.1 represents the number of operations that can be executed per clock cycle per SM. This is related with the number of

Table 2.1: GPU technical specifications, according to their Compute Capability.

Technical Specifications	Compute Capability						
	1.0	1.2	2.0	3.0	3.5	5.0	5.2
Warp shuffle functions	No			Yes			
Throughput of 32-bit floating-point operations*	8		32	192		128	
CUDA cores for arithmetic operations per SM	8		32	192		128	
Number of warp schedulers per SM	1		2	4			
Maximum number of threads per ThB	1024						
Warp size (in threads)	32						
Maximum number of resident ThBs per SM	8		16			32	
Maximum number of resident warps per SM	24	32	48	64			
Maximum number of resident threads per SM	768	1024	1536	2048			
Number of 32-bit registers per SM	8 K	16 K	32 K	64 K			
Maximum amount of shared memory per SM	16 KB		48 KB			64 KB	96 KB
Amount of local memory per thread	16 KB		512 KB				
Constant memory size	64 KB						
Number of shared memory banks	32						

*Add, multiply, multiply-add.

cores per SM. Hence, the values in Table 2.1 should be multiplied by the number of existing SMs, in order to get the actual throughput for the whole device. For a warp size of 32, one instruction corresponds to 32 operations, so if N is the number of operations per clock cycle, the instruction throughput is $N/32$ instructions per clock cycle.

Table 2.1 also includes the maximum number of resident ThBs, warps and threads per SM, for the different compute capabilities. Similarly, the memory characteristics, such as the number of 32-bit registers per SM, the maximum amount of shared memory, the amount of local memory per thread and the constant memory size were also presented.

The shared memory is divided into equally-sized 32 memory modules, denoted as *banks*, which can be simultaneously accessed to achieve high bandwidth. Any memory request (read/write) made of x addresses that fall in x distinct memory banks can therefore be serviced in parallel, yielding an overall bandwidth that is x times as high as the bandwidth of a single module. However, if two addresses of a memory request fall in the same memory bank, there is a *bank conflict*, the accesses have to be serialized, thus decreasing the effective bandwidth. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, thus decreasing the throughput by a factor equal to the number of separate memory requests.

Table 2.2 presents a summary of the device memory features, which includes for each memory space: *i*) location (on/off chip); *ii*) whether it is cached; *iii*) how it is accessed, i.e., read/write (R/W); *iv*) visibility; and *v*) the lifetime of the memory space.

2.4 CUDA Alternatives for Extended Portability: OpenCL

The information that was provided for the NVIDIA programming model (i.e., CUDA), as well as the terminology associated to those GPU architectures is easily extended to other frameworks,

Table 2.2: Device memory features summary.

Memory Space	Location	Cached	Access	Visibility	Lifetime
Register	On chip	n/a	R/W	1 thread	Thread
Local	Off chip	Yes*	R/W	1 thread	Thread
Shared	On chip	n/a	R/W	All threads in ThB	ThB
Global	Off chip	**	R/W	All threads + host	Host
Constant	Off chip	Yes	R	All threads + host	Host
Texture	Off chip	Yes	R	All threads + host	Host

*Cached in L1 and L2 by default on devices of compute capability 2.x and 3.x; devices of compute capability 5.x cache locals only in L2.

**Cached in L1 and L2 by default on devices of compute capability 2.x; cached only in L2 by default on devices of higher compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

such as OpenCL. As stated before, both CUDA and OpenCL provide similar functionality, where the OpenCL framework allows code portability among different devices.

In order to allow such aimed portability among different devices and architectures, OpenCL code is compiled at runtime. Nevertheless, the programming model and the assumed memory hierarchy are similar to CUDA, which have been presented in Sections 2.1 and 2.2. Table 2.3 presents a comparison between the terminology used in CUDA and OpenCL. For example, thread, ThB, shared memory and local memory in CUDA are denoted as work-item, work-group, local memory and private memory in OpenCL, respectively.

Nevertheless, since OpenCL is a language capable of being executed across several different platforms (e.g. CPUs, GPUs, Digital Signal Processors (DSPs) and FPGAs among others), the capabilities made available to current massively parallel computational hardware (such as GPUs) are sometimes limited. As an example, in some NVIDIA GPUs, where the L1 cache and shared memory use the same hardware resources, it is possible to configure the amount of shared memory per SM per kernel by using CUDA. This feature allows to fine tune the kernels that could achieve higher performance (e.g., in kernels which do not require shared memory, could take advantage of a larger L1 cache). OpenCL does not support such architecture-specific settings.

Table 2.3: Comparison between the terminology used in CUDA and OpenCL.

CUDA Terminology	OpenCL Terminology
Thread	Work-item
Thread Block (ThB)	Work-group
Global Memory	Global Memory
Constant Memory	Constant Memory
Shared Memory	Local Memory
Local Memory	Private Memory

2.5 Summary

In this chapter, a brief overview about modern GPU architectures, namely their programming model and memory hierarchy, was presented. Moreover, the architectural features of NVIDIA GPUs have been briefly discussed in order to provide the required information for supporting the remaining chapters of this Thesis. In particular, the kernels designed in Chapter 4 for the GHEVC video decoder shall maximize the *parallelism degree*, *instruction throughput* and optimize *memory accesses* [44].

3

Background and State of the Art

Contents

In this chapter, the background on HEVC is provided, together with the state-of-the-art on HEVC decoder implementations.

3.1 General Overview of the HEVC Standard

According to the HEVC standard, a video frame is decoded from the received bitstream in data element units corresponding to square pixel blocks. These pixel blocks are denoted as Coding Tree Units (CTUs) [45], whose size information ($N \times N$) is decoded from the received bitstream. Possible values for N are 64, 32 and 16 pixels [4]. Each CTU is further split into square blocks with $L \times L$ pixels, named Coding Units (CUs), by following a quadtree structure [5]. The dimension of the CUs (L) varies between a maximum size of N pixels to a minimum size of 8 pixels, as shown in Figure 3.1.

Each CU encloses a Prediction Unit (PU) and a Transform Unit (TU) [45], used for generating the prediction pixel block and the corresponding residual data, respectively. The prediction pixel block can be obtained either by using the data from the same frame (intra prediction) or from previously decoded frames (inter prediction). A general framework of an HEVC decoding structure, together with the corresponding module integration, is presented in Figure 3.2.

As defined by the standard, an HEVC decoder consists of the following main modules:

- **Entropy Decoder**: decodes the input bitstream and collects the required data to decompress the video sequence.
- **De-quantization and Inverse Transform (DIT)**: recovers the pixel residues by dequantizing the entropy decoded coefficients and by applying the inverse integer transforms to such coefficients, reverting them into the pixel domain, by considering up to four different block sizes and five integer discrete transforms [46].
- **Motion Compensation (MC)**: reverts the PU inter prediction, by considering the previously decoded frames as reference frames, as well as symmetric and asymmetric partitions, quarter-pel motion vectors, multiple reference frames and an interpolation procedure with up to 8-tap filters [47].

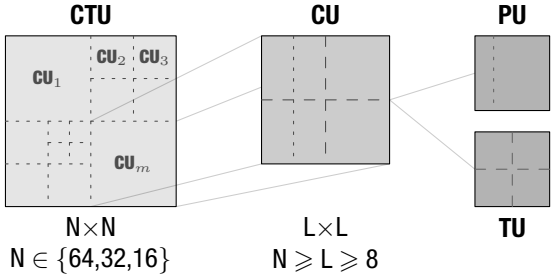


Figure 3.1: Example of the CTU partitioning into CUs, PUs and TUs.

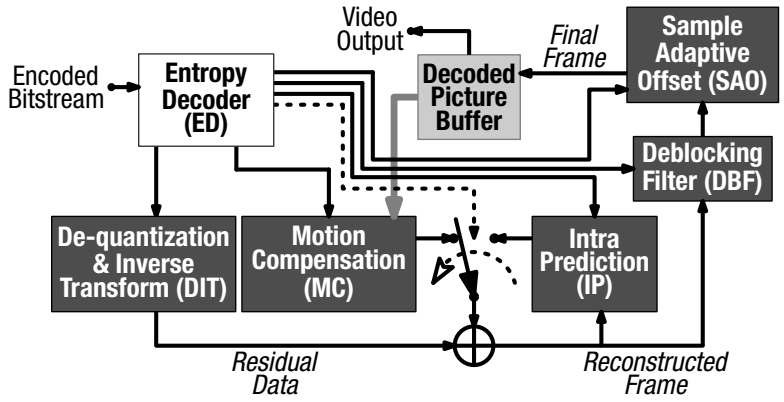


Figure 3.2: Block diagram of an HEVC decoder.

- **Intra Prediction (IP)**: executes the PU intra prediction, where the spatial prediction mechanism considers only the already reconstructed neighboring pixels of the current frame to predict the current block, which is subsequently added with the residual data computed by the DIT [48]. Several block sizes have to be considered, as well as thirty five different IP modes that are specified by the HEVC standard [48]. Since each prediction mode takes into account the reconstructed pixel samples of the neighboring blocks, it must respect strict data dependencies imposed by the HEVC standard. These dependencies do not only occur between adjacent data blocks within the IP module, but also across the DIT and IP modules;

- **Deblocking Filter (DBF)**: reduces the blocking artifacts of the reconstructed blocks from the hybrid video coding[1]. The DBF is applied to a 8×8 sample grid of the frame, where the vertical edges are processed first, followed by the horizontal ones [50].

- **Sample Adaptive Offset (SAO)**: improves the overall image quality, by reducing the CTU sample distortion according to a set of parameters selected at the encoder [51].

The decoding procedure, depicted in Figure 3.2, starts by decoding the *Encoded Bitstream*, using the *Entropy Decoder* module, in order to obtain the coefficients, as well as all other information required to decompress the video sequence. The coefficients are then de-quantized and inverse transformed by the DIT module, in order to obtain the residual data. Then, the reconstructed image blocks are obtained by adding the residual data from the DIT module to the predicted image blocks, computed either in the IP or in the MC modules. Then, DBF is applied to attenuate blocking artifacts introduced by the block-based prediction and transform coding. Finally, the mean sample distortion is reduced in the SAO module, where the final *Video output* is produced.

It is important to notice that the reconstruction process (DIT, MC and IP) is executed at block-level, where the CTUs are processed in raster-scan order. Inside each CTU, the CUs are decoded

[1] The hybrid video coding scheme combines temporal prediction between pictures of the video sequence with transform coding techniques for the prediction error [49].

by following a z-scan order, as well as the PUs and the TUs within each CU. On the other hand, the in-loop filters (DBF and SAO) are applied at frame-level on the reconstructed frame.

Although most of the HEVC procedures provide a high level of parallelism, the bottlenecks (in terms of parallelization) are the IP and the *Entropy Decoder*, as presented in [24].

3.1.1 Entropy Decoder

As defined by the HEVC standard, a video bitstream is a set of encoded syntax elements, which carry the information on how the video signal can be reconstructed at the decoder [52] (see Figure 3.3). These syntax elements are encoded with three coding schemes: fixed length codes, zero-order Exponential-Golomb code and arithmetic coding [53]. As a consequence of the rather efficient encoding success of the H.264/MPEG-4 AVC Context-Adaptive Binary Arithmetic Coding (CABAC) [54], a similar but still improved coding scheme was adopted in the HEVC standard, which improves the throughput, memory requirements and compression performance [55].

The CABAC coding engine requires the transformation of non-binary syntax elements into a binary representation before encoding. This process is called *binarization*, where each binary symbol is named as *bin*. Each *bin* is then coded according to its respective *probability model* (or *context model*), where the probabilities for the two possible binary values “0” and “1” are stored. The *context model* can be static, with uniform probability distribution, or adaptive, when the probability distribution is updated at each coded symbol. At the decoder side, the encoding process is reversed to reveal the coded syntax elements from the bitstream. Hence, the CABAC context adaptability guarantees high coding efficiency but also increases the data dependency and unpredictability of the *Entropy Decoder* module, which restricts parallel implementations.

Therefore, in order to increase the level of parallelism of the *Entropy Decoder* module, the CABAC data dependencies are reduced in the HEVC standard by three frame-level parallelization strategies: slices, tiles and Wavefront Parallel Processing (WPP) [56]. Each picture of the video

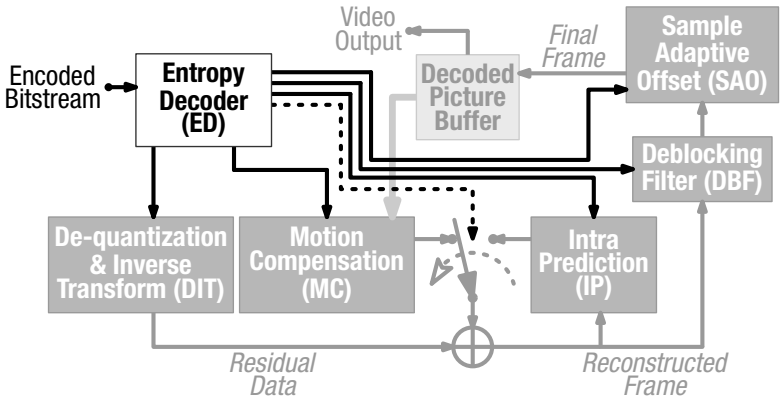


Figure 3.3: Block diagram of an HEVC decoder: entropy decoder.

sequence is divided into one or more slices, where slices are regions of the frame composed of an integer number of CTUs that can be independently decoded. However, although the entropy decoding and the reconstruction frame process of a slice may be performed independently from other slices of the frame, there are still potential dependencies regarding the cross-slice border in-loop filtering. There are three types of slices:

- **I Slice**: all PUs are intra predicted.
- **P Slice**: each PU may be intra or inter predicted, where the motion compensation is performed on a single reference frame from the reference picture list with a single PU motion vector.
- **B Slice**: each PU may be intra or inter predicted, where the motion compensation is performed with up to two reference frames from two reference picture lists.

Nevertheless, the coding efficiency usually decreases when the number of slices per frame increases, mainly due to the slice header overheads and reduced spatial redundancy exploitation.

On the other hand, a tile is a new video coding parallelization strategy defined by the HEVC standard. A tile is a rectangular region of the frame that can be independently decoded from the other tiles of the frame [57], but which provides more parallelism to be exploited by parallel architectures. However, although the entropy decoding and the reconstruction frame processing of a tile are performed independently from other tiles of the frame, the in-loop filtering can still be applied over tile boundaries to avoid tile border artifacts. Moreover, if multiple slices are employed with tiles, one of the following conditions shall be true per slice and per tile: *i*) all CTUs of a slice belong to the same tile or *ii*) all CTUs of a tile belong to the same slice. Hence, although the tile coding efficiency is higher than when multiple slices are used, it decreases when the number of tiles increases, which limits the parallelization efficiency [56].

Just like the tile strategy, the WPP has been introduced by the HEVC standard to take advantage of multi-threading architectures. Each CTU row of the frame can be decoded with a different thread. Nonetheless, to exploit statistical redundancy, each CTU of a frame can only be processed if the two consecutive and immediately above CTUs have been already decoded, which provides a wavefront approach. Although the WPP parallelism level is limited by the frame size, a new strategy based on WPP, called Overlapped Wavefront (OWF), has been introduced in [56], which improves the efficiency of the WPP, by overlapping the execution of consecutive frames.

3.1.2 De-quantization and Inverse Transform

Figure 3.4 highlights DIT procedure in the HEVC block diagram. Thsi module is responsible for recovering the *Residual Data* from the entropy decoded coefficients. As referred before, the TUs in each CU are split into smaller blocks (4×4, 8×8, 16×16 or 32×32) according to a quadtree structure [58]. These blocks are named as Transform Blocks (TBs), and each TU is

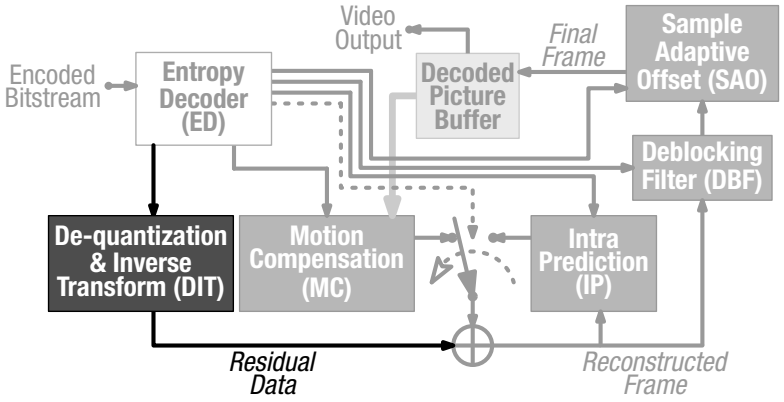


Figure 3.4: Block diagram of an HEVC decoder: de-quantization and inverse transform.

composed by one luma and two chroma TBs. Similarly to H.264/MPEG-4 AVC, only integer core transforms are specified by the HEVC standard, in order to avoid the introduction of rounding drifts, both at the encoder and the decoder, caused by rounding on floating point computation. As a consequence, the HEVC 4×4 to 32×32 transform operations are based on the integer Discrete Cosine Transform (DCT) [46]. The inverse DCT kernels are the same for luma and chroma TBs, except for the 4×4 luma TB of intra blocks, for which an integer inverse Discrete Sine Transform (DST) is applied.

The DIT procedure (see Figure 3.5) is directly applied on the *TB coefficients* obtained from the entropy decoder, in order to obtain the TB *Residual Data*. A more in-dept discussion about the details of transform coefficient coding [59] (such as coding methods for the last significant coefficient, significance map, coefficient levels and sign data) are beyond the scope of this thesis.

3.1.2.A Residual Data Decompressing

For each TB, the overall procedure is controlled by three flags [60]:

- **Transquant Bypass Flag (TBF)**: to indicate a bypass operation over the inverse transform and de-quantization procedures. This flag is encoded at CU-level, at the beginning of the CU syntax structure. Moreover, this flag enables a perfect reconstruction for a lossless representation of the coded block, since the residual signal is directly coded without any degradation [61, 62].

- **Coded Block Flag (CBF)**: indicates the presence of nonzero transform coefficients at the TB-level. If this flag is unset, the corresponding TB *Residual Data* is a null block.

- **Transform Skip Flag (TSF)**: to indicate a skipping of the inverse transform at the TB-level. Usually, this technique improves the compression efficiency of screen-content video sequences, which contains text and graphics [60].

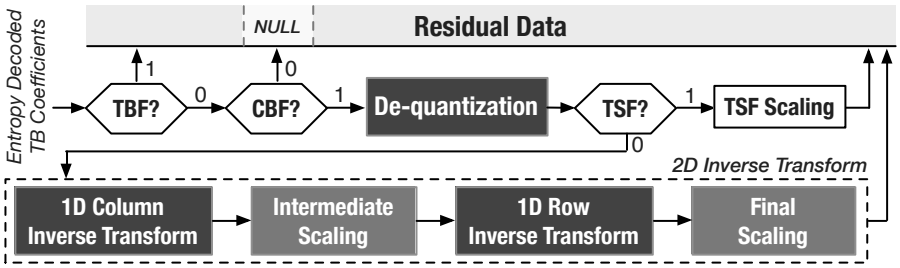


Figure 3.5: The HEVC residual data decompressing flowchart.

The overall procedure of the DIT module is presented in Figure 3.5. Whenever the TBF is set, DIT is bypassed, which means that the residual data directly corresponds to the TB coefficients. To achieve lossless CTU (or frame) reconstruction, the TBF can be used by the encoder to bypass not only the DIT module, but also the DBF and the SAO modules. If the TBF is unset and the CBF is set, the de-quantization procedure must be applied to the TB coefficients. The *De-quantization* module also implements the HEVC inverse scaling, which depends on the Quantization Parameter (QP) and on the adopted TB size [4]. Finally, the TSF signals the decoder to skip the inverse transform and to apply only the *TSF Scaling* (TSF=1). When TSF is not set, the *2D Inverse Transform* is performed on the de-quantized data block, by computing two 1D decompositions (i.e., *1D Column* and *1D Row Inverse Transforms*). Each decomposition is followed by a specific scaling procedure to perform the normalization in the transform domain [46]. Each 1D decomposition is performed on a specific *transform coefficient array*, which is chosen with respect to the adopted TB size and prediction mode.

The HEVC standard also defines the I_PCM mode, to reconstruct unusual blocks (e.g., noise-like pixel blocks) at the CU level [5]. In this case, the final block is obtained directly from the bitstream, without the application of the remaining HEVC procedures, such as DIT, prediction or in-loop filtering [62]. Due to the lossless representation of this mode, the amount of bits to encode a CU as I_PCM mode can be considered as an upper limit of the amount of bits required for encoding a CU. For rare noisy content, the encoder may switch to I_PCM mode.

3.1.2.B DIT Parallel Implementations

Regarding the DCT, a parallel implementation (using OpenCL) of the real (non-integer) DCT for image compression was already proposed in [63], by using a floating-point representation. Nevertheless, not only is such non-integer transform not compliant with most recent video standards, but the strict temporal requirements that are imposed in video coding are significantly more demanding than in image processing. In [64], the HEVC inverse transform was implemented in a GPU without considering the skipped and bypassed modes, as well as the memory transfers between the CPU and the GPU.

In what concerns the transform module, several algorithms were already proposed to alleviate

the complexity of the encoder side, like an implicit transform unit partitioning [65] and the zero block detection [66, 67], which eliminates redundant computations based on [68].

At the decoder side, parallel implementations often pose a difficult challenges, not only because the decoder should be able to support bitstreams produced by any encoder configuration, but also because the processing platform at the decoding device often exhibits restrictive processing capabilities. In [69], the NEON SIMD instruction set extension of an ARM processor is exploited to accelerate the transform and inverse transform procedures, in order to be applied in mobile and tablet devices, achieving a speed up of 5.6$\times$ over the reference software for Full HD sequences.

Several hardware implementations of the HEVC inverse transform procedure can also be found in the literature. For example, in [46, 70], a unified forward and inverse transform architecture is proposed to save circuit area in hardware implementations. Another architecture is provided in [71] which is able to process 8 coefficients/cycle using a 65 nm standard cell technology. In [72], a dedicated architecture was designed to process Ultra HD 4K at 30 frames/s in a 40 nm technology, with a throughput of 2 coefficients/cycle in the worst case scenario. Ultra HD 4K at 30 frames/s is also supported by [73], where a power consumption as low as 3.9 mW is obtained with 90 nm technology. In [74], 32 coefficients/cycle for any combination of TU sizes is proposed with a 45 nm technology.

3.1.3 Motion Compensation

Similarly to the previous video standards, the MC techniques adopted by HEVC aim to predict each pixel block by using information from temporal neighboring frames, also known as reference frames (see Figure 3.6). Those reference frames are stored in two picture buffers, i.e., List 0 and List 1, and must be exactly the same in both the encoder and the decoder, in order to obtain common predicted blocks [75].

On the decoder side, the MC is performed with the motion data encoded in the received bitstream, including: *i*) the pixel block size (PU partitioning); *ii*) the prediction direction, which defines the used picture buffers, i.e., List 0, List 1 or both; *iii*) the reference frame indexes, which specify the reference frames used in each list; and *iv*) the motion vectors, which define the displacement between the positions of the original block and the predicted block in the reference frame [76].

3.1.3.A PU Partitioning

The PU is divided into luma and chroma Prediction Blocks (PBs), and the MC is applied to each PB. The final reconstructed block is obtained by adding the prediction data from the PB and the residual data from DIT. In particular, when the usual 4:2:0 chroma subsampling is adopted, the chroma blocks are four times smaller than the corresponding luma blocks. Furthermore, when a CU is encoded using inter prediction, the corresponding PU is split into one, two or four PUs. In

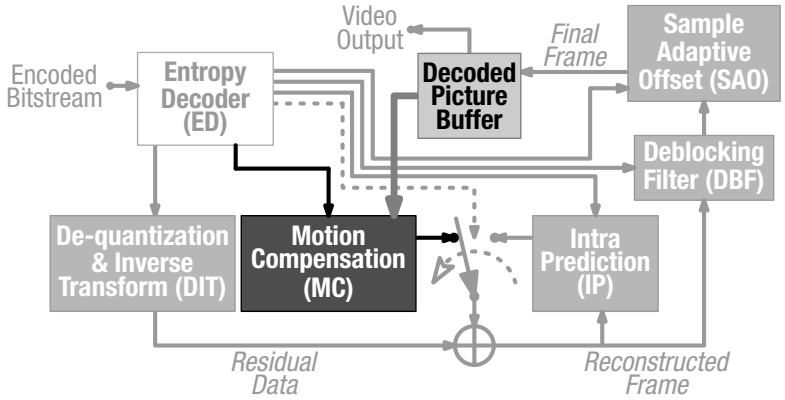
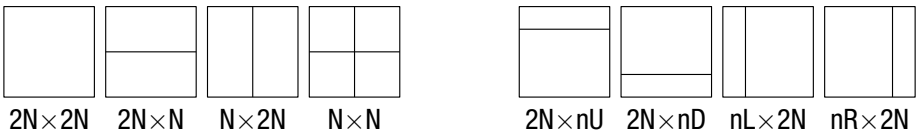


Figure 3.6: Block diagram of an HEVC decoder: motion compensation.



(a) Symmetric partitioning. **(b)** Asymmetric partitioning.

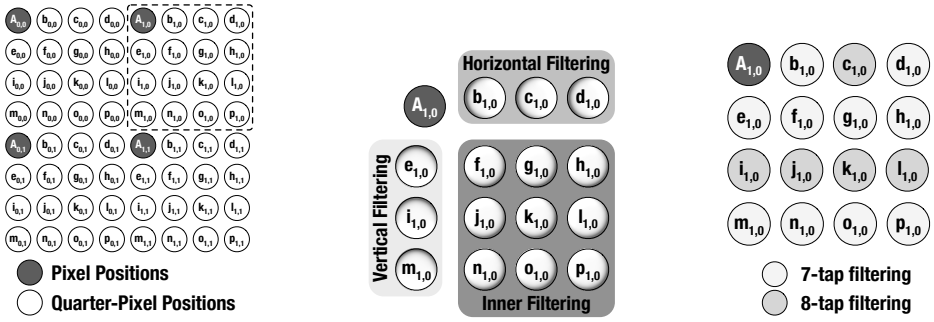
Figure 3.7: PU partition modes for the HEVC inter prediction.

Figure 3.7, all possible PU partition modes allowed are shown for the inter-coded CU, grouped in two subsets, i.e., *symmetric* and *asymmetric*.

For a $2N \times 2N$ CU, the *symmetric* partitioning is restricted to the quadtree structure, where a PU is split into up to four blocks (see Figure 3.7(a)). However, the PU can be divided into four blocks only if the CU could not be split into four CUs and the CU size is greater than 8×8 luma pixels. The HEVC standard also introduced *asymmetric* partition modes for Inter prediction (see Figure 3.7(b)). This added feature allows more accurate predictions and is responsible for up to 2.8% of bit-rate reduction [77]. Nevertheless, the *asymmetric* partition modes are unavailable when the CU size is equal to the minimum allowed size, in order to reduce the computational load. Thus, for an 8×8 CU, the possible PU partitions are 8×8, 8×4 and 4×8.

3.1.3.B Sub-pixel Interpolation

At the decoder, whenever the MC is performed within a single picture buffer (i.e., List 0 or List 1), the pixel samples of the PB are obtained by fetching a pixel block from the specified reference frame and picture buffer. The position of the pixel block is defined by the horizontal (x) and vertical (y) components of the motion vector. When the motion vector points to an integer pixel position (see $A_{x,y}$ in Figure 3.8(a)), the PB samples are directly obtained from the reference frame, i.e., no interpolation is performed. Otherwise, when the motion vector indicates a sub-pixel position, an interpolation procedure is started to obtain the fractional samples at positions from $b_{x,y}$ to $p_{x,y}$ (see Figure 3.8(a)) [47].



(a) Sample positions. **(b)** Filter directions. **(c)** Filter types.

Figure 3.8: Luma sample positions at quarter-pel resolution and filtering features.

In accordance just like the H.264/MPEG-4 AVC, the HEVC standard also specifies motion vectors at luma quarter-pixel resolution, but adopting different interpolation procedures. To generate such luma sub-pixel sample values, the HEVC standard defines three interpolation procedures: *Horizontal*, *Vertical*, and *Inner Filtering* (see Figure 3.8(b)). In the *Horizontal Filtering*, $b_{x,y}$, $c_{x,y}$ and $d_{x,y}$ samples are computed by filtering the pixels of the reference frame from the same row. In the *Vertical Filtering*, $e_{x,y}$, $i_{x,y}$ and $m_{x,y}$ samples are computed by considering the pixels in the same column of the reference frame. The samples produced by the *Inner Filtering* (see Figure 3.8(b)) are obtained by performing the vertical filtering on the samples from the same column, i.e., the previously produced sub-pixels $b_{x,y}$, $c_{x,y}$ or $d_{x,y}$ with *Horizontal Filtering*. For example, the *Inner Filtering* of $f_{x,y}$, $j_{x,y}$ or $n_{x,y}$ is performed by using $b_{x,y}$ samples. Hence, in *Inner Filtering*, the corresponding sub-pixel samples should be generated first with *Horizontal Filtering* and, only after, the vertical filtering should be applied.

For the luma component, the interpolation procedure is implemented by adopting 8-tap and 7-tap filters, according to each sub-pixel position. The 7-tap filtering is applied to create the sub-pixel samples that are close to the filtered samples, i.e., light gray filled sub-samples in Figure 3.8(c), while the remaining sub-samples are produced with 8-tap filtering. In what concerns the chroma interpolation, the filtering procedures are the same as for the luma component, but 4-tap filters are used.

When the MC procedure is performed by using both picture buffers (specified in the block prediction direction), the above-mentioned procedure is applied on both Lists in order to generate predicted blocks of each specified reference frame (one per List). Then, a particular set of weighted prediction parameters is applied on the obtained predicted blocks, in order to generate the final predicted block. These parameters, which are selected at the encoder side, are employed in a weighted arithmetic mean of the predicted blocks from both Lists. In the case where these parameters are not present in the bitstream, a simple average is performed instead.

3.1.3.C MC Implementations

By considering only the encoder side, most GPU-based implementations deal with the most computationally demanding procedure of the temporal prediction mechanism: the motion estimation (often also supporting relaxed dependencies), as proposed in [78, 79] for HEVC and in [80] for H.264/MPEG-4 AVC. At the decoder side, a GPU implementation of the H.264/MPEG-4 AVC interpolation module has been presented in [81], programmed in OpenCL and optimized to avoid performance penalties from the control and memory divergences.

In what concerns dedicated hardware, there are several HEVC interpolators implemented with different technologies. In [82], luma interpolation of Ultra HD 4K video sequences at 60 frames/s is performed with a 90 nm technology, running at 171 MHz. By also considering the chroma interpolation, a frame rate of 30 frames/s for Ultra HD 4K video sequences is achieved with a 150 nm technology in [83]. Interpolation filters for Ultra HD 8K video sequences at 60 frames/s are provided with a 40 nm technology in [84], and with a 65 nm technology in [85].

3.1.4 Intra Prediction

Unlike the MC, the IP procedure relies on the previously reconstructed blocks (see Figure 3.9). When a CU is encoded using intra prediction, the PU has the same size as the CU. The only exception occurs for the smallest CU size in the bitstream, where the PU can be further partitioned in four blocks (e.g., four 4×4 PUs for an 8×8 CU), which are processed in z-scan order [86].

3.1.4.A Dependencies

Similarly to the TU, the PU is further divided into luma and chroma PBs, and the intra prediction is applied to each PB. Just like in the inter-predicted PUs, the final reconstructed block is obtained by adding the prediction data from the PB and the residual data obtained from DIT. The PBs are processed in a strictly defined order, as specified by the HEVC standard [48]. The dashed-line

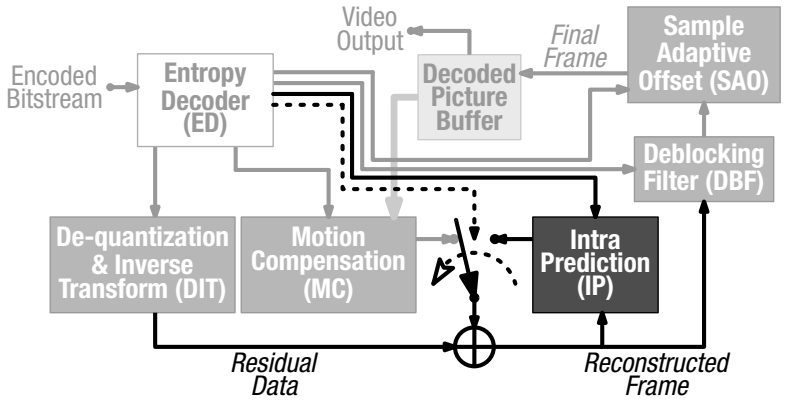


Figure 3.9: Block diagram of an HEVC decoder: intra prediction.

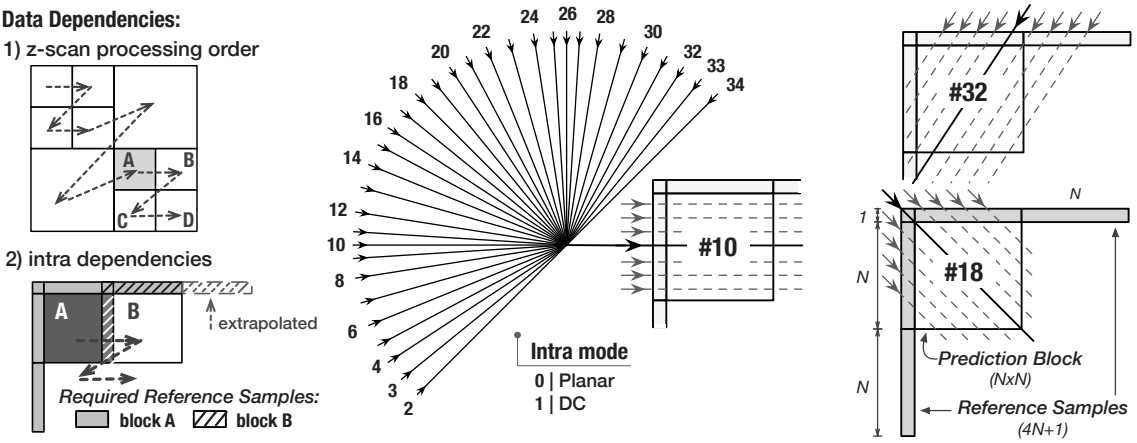


Figure 3.10: Intra prediction dependencies and modes.

in Figure 3.10 (in *Data Dependencies*) represents the data dependencies arisen from the *z-scan processing order* for different PUs. These *intra dependencies* occur because the previously reconstructed blocks (*reference samples*) are used as input for the next PBs.

Figure 3.10 also presents an example of the required references samples for blocks A and B (see *intra dependencies*). Block A requires the reconstructed data from adjacent upper and left blocks, while block B can only be predicted after the reconstruction of the reference samples from block A. In general, to perform the intra prediction of an $(N \times N)$ PB, $4N+1$ reference samples are required from up and left adjacent blocks. However, if the flag *constrained_intra_pred_flag* is set, only intra predicted reference samples can be used. In this case, the inter predicted reference samples are marked as not available.

Moreover, depending on the relative position of the PB in the CTU, one or more reference sample sets may not be available (see block B in Figure 3.10 in *2) intra dependencies*). If those reference samples lie outside the frame or belong to another slice, they are marked as not available. Whenever reference samples are not available, the remaining samples are *extrapolated* to fill the whole set of $4N+1$ samples, by repeating the value of the nearest available reference sample. If all reference samples are marked as not available, the $4N+1$ reference sample set is filled with the middle pixel value of the dynamic pixel range (e.g., 128 for 8-bit pixel values).

3.1.4.B Prediction Procedure

As soon as all the required $4N+1$ reference samples are generated, the intra prediction of the current PB can be started. For luma PBs, a smoothing filtering (low-pass filter) is firstly applied to the reference samples, according to the PB size and prediction mode. The HEVC standard also defines a strong filtering that can be applied to the reference samples of 32×32 luma PBs. After the smoothing stage, the reference samples are ready to be employed in the intra prediction procedure.

As presented in Figure 3.10, there are 35 different *Intra modes*: *i*) mode 0 refers to planar intra prediction; *ii*) mode 1 to DC prediction; and *iii*) modes 2 to 34 to angular predictions [87]. In the angular prediction mode, the interpolation is applied on the reference samples according to the specified direction [48] (e.g., #18 in Figure 3.10). Accordingly, each PB is predicted by using one of those intra modes, which can differ for luma and chroma PBs. When the TB is smaller than the PB, the intra prediction is performed at the TB level. In this case, each sub-block in the PB is predicted in z-scan order, where the size of each sub-block is defined by the TB. For intra prediction, TU can not be larger than PU [48].

3.1.4.C IP Implementations

Due to the intrinsic data dependencies and the low level of parallelism, the HEVC IP procedure is one of the hardest procedures to be parallelized, making difficult to be efficiently implemented in parallel architectures. Hence, only a few proposals which exclusively target the HEVC IP module implementation have been found in the literature for dedicated or reconfigurable hardware. In [88], an architecture for intra angular prediction of 4×4 and 8×8 blocks is proposed, which saves redundant computations to reduce the processing cycles and the consumed power in FPGA. A 40 nm technology has been exploited to process the IP module in [89], for which Ultra HD 4K at 30 frames/s can be achieved with 200 MHz. A throughput of 120 frames/s for the IP module is achieved for Ultra HD 8K video sequences with a 90 nm technology at 397 MHz in [90].

3.1.5 Deblocking Filter

In the HEVC standard, the DBF is performed after the *Reconstructed Frame* is obtained (see Figure 3.11). This module is applied to the boundaries of PBs or TBs, which rely on 8×8 samples grid for both luma and chroma [91].

3.1.5.A Boundary Filtering Strength

For each boundary, a Boundary filtering Strength (BS) is evaluated, according to several conditions from the neighboring blocks (see Table 3.1). The resulting BS value varies between 0 and 2, where 0 means that no deblocking filter will be applied. Whenever one of the neighboring blocks is intra-predicted, the BS value is always set to 2. Moreover, the chroma samples are only filtered when the BS value is 2 [50].

3.1.5.B Boundary Filtering

For luma, additional conditions are verified to determine whether the DBF should be applied [92, 93]. Each condition is verified for each set of 8×4 or 4×8 pixels, corresponding to the vertical and horizontal edges, respectively (see *Boundary Types* in Figure 3.12). Accordingly, a set of pixels in the first and the last row (or column) are used to decide which filter is going to be applied,

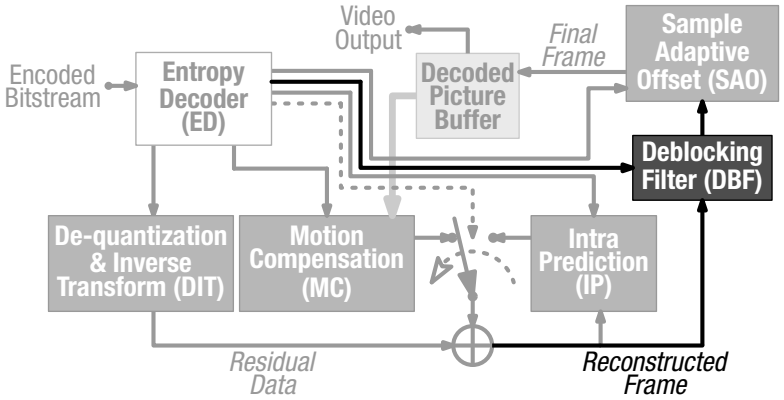


Figure 3.11: Block diagram of an HEVC decoder: deblocking filter.

Table 3.1: BS values for the luma boundary between two neighboring pixel blocks.

Boundary conditions	**BS**
One of the blocks is intra predicted	2
One of the blocks has residual data *and* it is a TU boundary	1
Different reference frames or number of motion vectors, between PUs	1
Absolute difference of corresponding motion vector component $\geq$1 pixel, between PUs	1
Otherwise	0

i.e., none, normal or strong (see dark-filled pixels in Figure 3.12). On each side of the boundary, only up to four neighboring samples have to be considered and up to three may be modified. For example, for the luma component, the strong filtering is applied on three pixels on each side of the boundary, while in the normal filtering at most two pixels can be filtered on each side of the boundary, depending on a set of DBF conditions (see *Strong Filtering* and *Normal Filtering* in Figure 3.12). In contrast, for chroma samples, the normal filtering is only applied on a single pixel on each side of the boundary.

It is worth noting that in the DBF overall process (as defined by the standard), all vertical edges from the frame are filtered before the horizontal edges [4]. However, as also defined by the standard, if the flag *pcm_loop_filter_disabled_flag* is set, the in-loop filtering procedures (DBF and SAO) are disable for the I_PCM predicted CU samples. Moreover, the samples that belong to a CU where TBF is set (lossless mode) should not be filtered as well.

3.1.5.C DBF Implementations

In [94], three different implementations are proposed for the HEVC DBF on multi-core systems. One approach divides the frame horizontally (for vertical filtering) and vertically (for horizontal filtering). The other implementations combine the vertical and horizontal filtering in a single pass, while the frame is vertically divided among CPU cores. When executed in a 6-core CPU platform, the best implementation of the proposed algorithms can obtain an average speedup of 5$\times$ for Full HD video sequences over a single core implementation. In [95], a better distribution of the

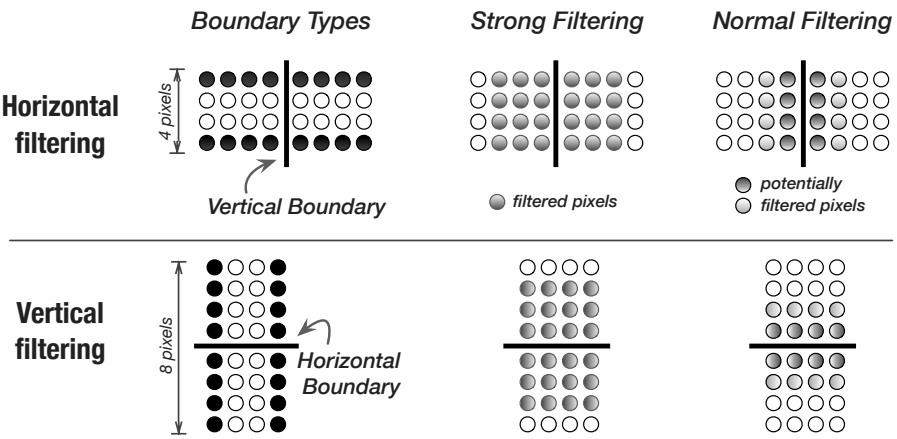


Figure 3.12: Deblocking Filter boundary and filtering types.

DBF computational load among the CPU cores is proposed by estimating the computational load at the CTU-level based on the CU depth (provided that the DBF is enabled).

A GPU implementation of the DBF has been proposed in [96]. An average frame processing time of 33 ms was achieved for Full HD video sequences on an NVIDIA GeForce 710M GPU. Another GPU-based implementation has been presented in [97]. An average frame processing time lower than 0.5 ms was achieved for Full HD video sequences with a NVIDIA Tesla K20M GPU.

An FPGA implementation of the HEVC DBF can process Full HD videos at 86 frames/s in [98]. By considering a video resolution of 4096×2048 pixels, DBF implementations can achieve 60 frames/s in [99] and in [100], with a 130 nm and a 45 nm CMOS technology, respectively. In [101], a DBF implementation for Ultra HD 8K video sequences can handle up to 123 frame/sec.

3.1.6 Sample Adaptive Offset

The deblocked samples are subsequently modified in the SAO module (see Figure 3.13), by adding an offset value according to a set of SAO parameters, namely: *Type*, *Offset Values* and *Band Position*/*Edge Class* [51]. These SAO parameters are encoded in the bitstream for each CTU and can have different values for luma and both chroma components, even in the same CTU [91]. The *SAO Type* parameter signals the decoder which SAO filtering should be applied (none, band offset or edge offset).

3.1.6.A Filtering Modes

In the band offset mode, the full amplitude of the pixel range is divided by 32 to define a set of *bands*. From this set, only four consecutive bands are considered for SAO filtering, according to the information stored in the bitstream (i.e., the *SAO Band Position* parameter). For each specified band, a single offset value is provided in the respective *SAO Offset Value* parameter. Then, all samples whose values belong to these four bands have to be modified, such that the deblocked

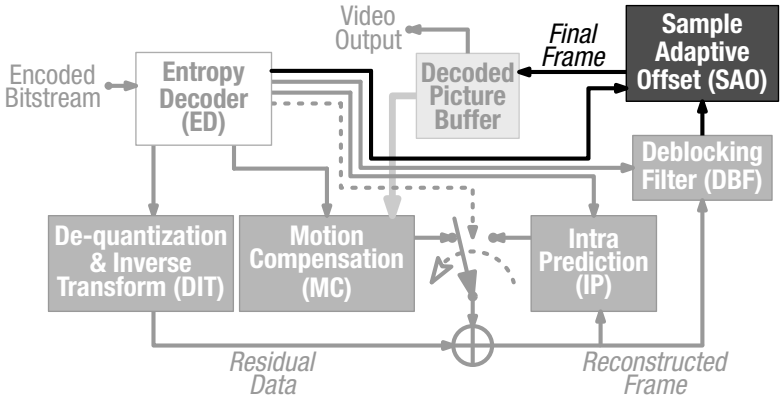


Figure 3.13: Block diagram of an HEVC decoder: sample adaptive offset.

sample value is added with the corresponding *SAO Offset Value*.

In the edge offset mode, the CTU samples are classified into four categories, according to the corresponding gradient direction, which is specified in the *SAO Edge Class* parameter. Figure 3.14 depicts all four possible gradient directions and allowed SAO categories. The gradient direction takes into account the pixel to be filtered represented by a ● symbol and two neighboring pixels depicted as the □ and the ○ symbols. The category is selected according to the pixel sample value differences between the three pixel, e.g., in *Category 1*, both pixel sample values of the neighboring pixels are higher than the one of the pixel to be filtered (see Figure 3.14). Similarly to the band offset mode, the offset value for each category is stored in the *SAO Offset Value* parameter. The *SAO Offset Value* is positive for categories 1 and 2, and negative for categories 3 and 4 (see arrow in Figure 3.14). Hence, whenever a pixel is classified in one of these categories, its deblocked sample is added to the corresponding *SAO Offset Value*. In contrast, the SAO is not applied if the samples are not classified in any of these categories.

The whole in-loop filtering process (DBF and SAO) is bypassed for the lossless mode (TBF=1) or when the samples are from an I_PCM predicted PU and the *pcm_loop_filter_disabled_flag* is set.

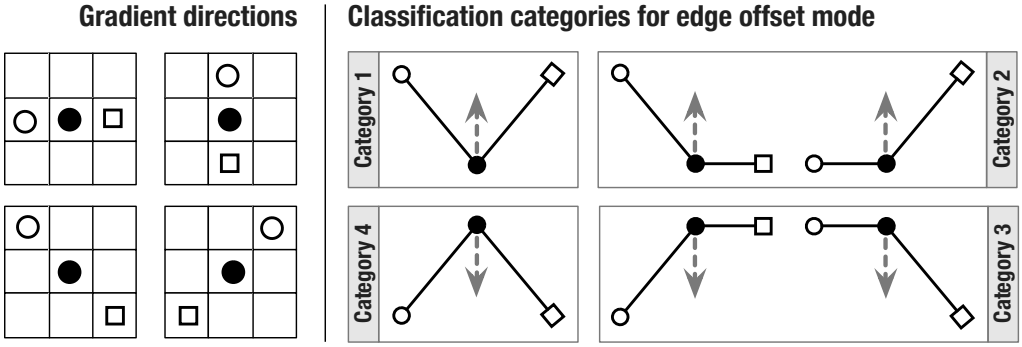


Figure 3.14: SAO gradient directions and classification categories.

3.1.6.B SAO Implementations

Most GPU-based HEVC SAO implementations are focused on the encoder side [102–104]. On the decoder side, a hardware-based HEVC SAO design is presented in [105], implemented with a CMOS 180 nm technology, where a CTU can be processed in 64 clock cycles.

When considering both the DBF and SAO, an HEVC in-loop filter implementation is proposed in [106] for low power programmable coprocessors, with luma frames of Full HD video sequences filtered with a 90 nm CMOS technology at 300 MHz and 25 frames/s. The design was further optimized in [107], where intra luma frames with Full HD resolution can be filtered at 152 frames/s using a CMOS 28 nm technology at 1.2 GHz. A hardware-based implementation capable of processing Ultra HD 4K at 60 frames/s is presented in [108] using a CMOS 28 nm technology running at 200 MHz. In [109], the in-loop filtering of Ultra HD 8K frames is performed at 120 frames/s on a CMOS 65 nm technology at 240 MHz.

3.1.7 Profiles, Tiers, and Levels

As mentioned before, a fully compliant HEVC decoder has to be able to decompress any compliant bitstream, for the defined set of profiles, levels and tiers [18], regardless of the involved computational complexity. The specified profile, level and tier of an HEVC decoder defines the required processing capabilities to decode a bitstream. A profile identifies the set of coding tools which may be used to encode a video sequence into a bitstream. However, the encoder may select a subset of the profile coding tools which considers to be suitable. The first version of the HEVC standard included three profiles: *Main*, *Main 10* and *Main Still Picture*. In those profiles, only 4:2:0 chroma subsampling was supported. The *Main* profile allows a bit depth of 8 bits per sample, while the *Main 10* profile allows bit depths of 8 to 10 bits per sample. The *Main Still Picture* profile is used to encode a single frame with bit depth of 8 bits per sample. Additionally, the *Range Extensions* [110] of the HEVC standard specifies 21 profiles which provide: *i*) support to 4:0:0, 4:2:2, and 4:4:4 chroma subsampling formats; and *ii*) increased sample bit depths beyond 10 bits per sample.

The set of restrictions on the parameters that determine the decoding and buffering capabilities are indicated by the levels. These parameters include: maximum picture size, the coded and decoded picture buffer sizes, the maximum number of slice segments and tiles in a picture, as well as the maximum sample rate and maximum bitrate. In Table 3.2, it is presented some of the limits specified by the standard according to the considered level, for the *Main* and the *Main 10* profiles. Furthermore, in order to provide different bitrate ranges across consumer and professional applications, the HEVC standard specifies two tiers named as *Main* and *High*. For example, the maximum bit rate (in kbit/s) for the *Main* and the *Main 10* profiles are presented in Table 3.2. It is important to notice that any decoder compliant with some specific level can still perform higher rates, but never lower. As it can be noticed in Table 3.2, an HEVC decoder compliant with level 5

Table 3.2: Tier and level limit examples for *Main* and *Main 10* profiles.

Level	Maximum luma picture size (samples)	Maximum luma sample rate (samples/sec)	Maximum bit rate (kbit/s) *Main tier*	Maximum bit rate (kbit/s) *High tier*	Ultra HD 4K frame rate (frames/s)
1	36 864	552 960	128	-	-
2	122 880	3 686 400	1 500	-	-
2.1	245 760	7 372 800	3 000	-	-
3	552 960	16 588 800	6 000	-	-
3.1	983 040	33 177 600	10 000	-	-
4	2 228 224	66 846 720	12 000	30 000	-
4.1	2 228 224	133 693 440	20 000	50 000	-
5	8 912 896	267 386 880	25 000	100 000	32
5.1	8 912 896	534 773 760	40 000	160 000	64
5.2	8 912 896	1 069 547 520	60 000	240 000	128
6	35 651 584	1 069 547 520	60 000	240 000	128
6.1	35 651 584	2 139 095 040	120 000	480 000	256
6.2	35 651 584	4 278 190 080	240 000	800 000	300

should not process an Ultra HD 4K video sequence with less than 32 frames/s.

3.2 Complete HEVC Decoder Implementations

The previously mentioned publications take into consideration only one or two HEVC procedures. In contrast, the following works provide details on how to efficiently implement the whole HEVC decoder on different architectures. When considering CPU-based decoders, the HEVC Test Model (HM) reference software [111] is still the most used decoder for research purposes. However, it is not optimized for practical applications neither does it target real-time performance. As a consequence, the open-source OpenHEVC [112] decoder, heavily optimized for SIMD vectorization, is usually regarded as a more suitable benchmark and it will be herein adopted for the performance evaluation in Chapter 5.

3.2.1 CPU-based Implementations

In [16], SIMD parallelization models at the level of HEVC decoder modules are exploited by specifically focusing on current multi-core CPU architectures. At the end, the SIMD-based decoder implementation has been able to decode Full HD video sequences at 133 frames/s on average using only one core of an Intel i7-4770S processor, operating at 3.1 GHz. However, when employing the OWF method proposed in [56], the proposed HEVC decoder is able to reach a performance of 543 frames/s with 8 CPU threads for Full HD video sequences.

In [113], a SIMD HEVC decoder implementation is proposed, which showed to be able to decode 40 Full HD video frames/s on the Intel i5-2400 processor. Although this decoder is 4$\times$ faster than version 4.0 of the reference software HM, a multi-core approach of the same decoder provides speedups up to 13.2$\times$ with 4-thread parallel decoding over a single Intel i5-2400 core

processor [114]. In [115], another multi-core approach achieved a speedup up of 2.9× with six threads compared to the HM 12.0 decoder, when using an Intel Core i7-3960X processor at 3.3 GHz. An HEVC decoder is also proposed in [116], where a task graph is built by restructuring the sequential part of the decoder. In [117], a hybrid parallelization strategy of the HEVC decoder, by combining task-level parallelism and data-level parallelism on CTUs is proposed. Experimental results show that Full HD video sequences can be decoded at 78 frames/s for a QP value of 29 in an Intel Core i7-3770k, running at 3.5 GHz.

In [19], the Lentoid HEVC/H.265 Decoder (LentoidDec), developed by Strongene Ltd is presented. SIMD instructions were also exploited to speed up the LentoiDec decoder, where a frame rate of 40-75 frames/s was obtained for Ultra HD 4K videos on an Intel i7-2600 with 3.4GHz quad-core processor and four decoding threads. Moreover, by only employing 2 threads, a frame rate of 35-55 frames/s is still achieved for HD videos on an ARM Cortex-A9 duo-core processor executing at 1.2 GHz.

When considering embedded processors, a frame rate greater than 30 frames/s is obtained for HD videos in [118], by also exploiting SIMD instructions with an ARM Cortex-A9 duo-core processor performing at 1.2 GHz. In [119], the processor low power states and code optimization techniques are studied to achieve a better power efficiency of an HEVC decoder. Experimental results show that in a real-time decoding scenario, lowering the frequency and using more cores can provide a lower energy consumption than finishing faster in order to become idle longer, by considering the power consumption of the CPU, memory and “uncore” modules. A Dynamic Voltage Frequency Scaling (DVFS) low power consumption software architecture for the HEVC decoder is also presented in [120]. Another low power and architecture-aware implementation of the HEVC decoder is presented in [121]. The scheduling strategy tuned for a Samsung Exynos 5422 SoC achieves 24 frames/s for Full HD video sequences and is still able to reduce the overall energy consumption by 20%. In [122], it is presented an analysis of possible issues of an HEVC decoder implementation for heterogeneous embedded systems, by considering both the CPU and the GPU.

3.2.2 Dedicated Architectures Implementations

In [123], a DSP-based HEVC decoder is presented by employing Reconfigurable Video Coding CAL Actor Language (RVC-CAL). A multi-core approach of the HEVC RVC-CAL decoder with Open Multi-Processing (OpenMP) API is provided in [124], where up to 4 cores have been used.

When FPGA implementations are considered, a frame rate of 30 frames/s of Full HD video resolution is obtained in [125] with a 65 nm technology. By relying on a 40 nm CMOS technology, a frame rate of 30 frames/s for Ultra HD 4K video sequences is attained in [126]. In [127], 60 frames/s (also for Ultra HD 4K video sequences) are achieved with a 28 nm CMOS technology. Another FPGA-based HEVC decoder based on a 28 nm technology is presented in [128], where the design

is estimated to be able to decode Ultra HD 4K videos at 60 frames/s, on a Xilinx Virtex-7 FPGA XC7V2000T at 400 MHz with a dual-core processor or at 200 MHz with a quad-core. In [129], a two-stage subpipelining scheme to reduce on-chip SRAM is presented for an HEVC video decoder application specific integrated circuit. Ultra HD 4K video sequences can be decoded at 30 frames/s, with a 1.77 mm^2 circuit area for a 40 nm CMOS technology. A hardware-based HEVC decoder with support to the Main 10 profile is presented in [130], which is able to decode Ultra HD 4K video sequences at 60 frames/s. By exploiting a 90 nm CMOS technology, the HEVC decoder proposed in [131] can decode Ultra HD 4K video sequences at 30 frames/s with a 270 MHz operating frequency. Another FPGA implementation of the HEVC decoder is presented in [132], targeting 30 frames/s of Ultra HD 4K video sequences on a Xilinx Zynq 7045 with an operating frequency of 150 MHz.

3.2.3 GPU-based Implementations

When considering GPU-accelerated HEVC decoders, it is observed that most commercial applications take advantage of the dedicated hardware structures inside the GPU to perform video decoding. However, it is worth noticing that most hardware-based HEVC decoders on current GPU devices are only available for certain types of architectures, and they are also limited to certain HEVC profiles (e.g., in the NVIDIA GM206 architecture, it is only possible to decode the Main profile up to Level 5.1 [133]). In this scenario, the dedicated hardware is usually accessed through a specific API (such as the Microsoft DirectX Video Acceleration[2]) and implemented in the wrapping software (e.g., LAV Filters[3]).

Regarding software video decoding on GPUs, OpenCL has been also used to provide HEVC decoding capabilities in several commercial decoders. For example, the Ittiam's i265[4] family of products includes OpenCL-based HEVC decoders for Intel HD Graphics, Iris and Iris Pro GPUs, AMD GPUs, among others. Another OpenCL-based HEVC decoder for AMD GPUs is provided by the Strongene OpenCL H.265/HEVC Decoder for Windows[5]. CyberLink PowerDVD also provides OpenCL-based HEVC decoder capabilities[6]. However, a direct comparison with these commercial applications is difficult to make, due to the impossibility to decouple the segments that strictly deal with the GPU-based decoding. In fact, most of these commercial solutions provide a very deep integration of these routines in a more general application and the implementation details are either not disclosed or the source codes are not publicly available.

[2]\[Online] Available: `https://msdn.microsoft.com/en-us/library/aa965263.aspx` [Accessed 01 Dec. 2016]
[3]\[Online] Available: `https://github.com/Nevcairiel/LAVFilters` [Accessed 01 Dec. 2016]
[4]\[Online] Available: `http://www.ittiam.com/products/software-ips/video/h265-hevc` [Accessed 01 Dec. 2016]
[5]\[Online] Available: `http://www.strongene.com/en/downloads/downloadCenter.jsp` [Accessed 01 Dec. 2016]
[6]\[Online] Available: `http://www.cyberlink.com/products/powerdvd-ultra/features_en_EU.html` [Accessed 01 Dec. 2016]

3.3 Summary

This chapter presented an overview of the state-of-the-art HEVC standard in what concerns its decompression procedures, from the entropy decoded syntax elements down to the list of decoded video frames. It presented a brief explanation about the HEVC entropy decoder, together with the frame-level parallelism provided by the standard. Moreover, a detailed explanation about the decompressing procedures applied to the received syntax elements was presented, namely the: DIT, MC, IP, DBF and SAO. Furthermore, state-of-the-art implementations for each decoding procedure were discussed. Finally, complete HEVC decoder implementations referred in the literature were also described, including several commercial frameworks, which represent the current state-of-the-art on HEVC decoding.

The set of concepts that were presented herein will be further discussed throughout this dissertation. In particular, the proposed GPU approach of each HEVC decoding procedure, which aims to exploit the maximum level of parallelism, is presented in the next chapter.

4

GHEVC Parallel Algorithms

Contents

The proposed GHEVC decoder is supported on a heterogeneous platform composed by a CPU and a GPU. The CPU is responsible for the entropy decoding and for orchestrating and ensuring the correct execution order of the GPU kernels, including the data transfers to and from the GPU memory. In the proposed GHEVC decoder, only the HEVC decoding modules (i.e., DIT, MC, IP, DBF and SAO) are designed to efficiently exploit the capabilities of highly parallel GPU architectures. They leverage the fine-grain parallelism of these computationally complex and highly data dependent modules, while providing fully compliant HEVC decoding. To achieve the aimed performance, the proposed algorithms maximize the number of active warps, while ensuring that all threads in a warp perform the same operation from a kernel. Furthermore, data accesses are carefully managed, in order to efficiently use the complex GPU memory hierarchy, i.e., global, cache, shared, constant and texture memories [36].

The GPU constant and texture memory spaces are mostly used to store information that does not change along the video sequence decoding process. In particular, the GPU texture memory is used to save the *Transform Coefficient Arrays* (see Chapter 3), whereas the constant memory is used to store:

- **Frame height and width** – the frame size is employed mainly in the thread and warp positioning inside of each frame;
- **HEVC tables** – tables specified by the HEVC standard, such as *intraPredAngle* and *invAngle*, used for the angular intra prediction;
- **HEVC filter coefficients** – the interpolation filter coefficients of the luma 8-tap and the chroma 4-tap filters of the *Motion Compensation* kernel;
- **HEVC flags** – the HEVC control flags (e.g., *pcm_loop_filter_disabled_flag*), which specifies modules behavior;
- **List 0 and 1** – the reference frames, used in the *Motion Compensation* kernel, which are stored in the GPU global memory. However, since the same reference frame can be in both lists at the same time, *List 0* and *1* are created in the constant memory as arrays of pointers to the GPU global memory to avoid data replication.

It is worth noting that although some information is sent only once at the beginning of the decoding process, other parameters are updated more frequently (e.g., *List 0* and *1*). The remaining data that is needed for each kernel (i.e., DIT, MC, IP and SAO) are transferred to the GPU global memory before the execution of the respective GPU kernels.

4.1 Sequence-level and Frame-level Parallelism

In order to ensure a fully compliant and real-time HEVC decoding, the implemented parallel algorithms for the different HEVC modules are closely integrated into a collaborative CPU+GPU

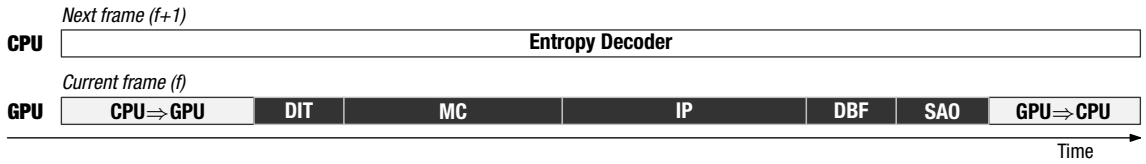


Figure 4.1: Proposed CPU+GPU integration of the GHEVC decoder.

decoding environment. In the proposed GHEVC decoder, the entropy decoder is the only HEVC module that is performed on the CPU, due to its highly sequential and irregular nature, while all remaining HEVC modules are implemented on the GPU, by relying on the proposed parallelization approaches (see Figure 4.1).

The collaborative CPU+GPU HEVC decoder starts by acquiring the bitstream portion that corresponds to a frame and by performing the entropy decoding on the CPU side. Then, the entropy decoded data is sent to the GPU and used as the input data for the subsequent decoding (see "*CPU⇒GPU*", in Figure 4.1). The first kernel to be executed corresponds to the DIT module, in order to compute the residual data for the prediction kernels (MC and IP). Afterwards, the MC module is executed before the IP kernel, in order to produce the reconstructed blocks of the inter predicted CUs. After the IP module, the whole reconstructed frame is present in the GPU global memory and used as input for the DBF module. Although the DBF is performed "in place" over the reconstructed frame (to produce the deblocked frame), the SAO module is not an "in place" algorithm. In this way, to ensure compliance with the HEVC standard, all warps from the SAO module can only read the deblocked frame and write the final frame into a separated memory space. It is worth noting that while the deblocked frame memory space is reused to store the reconstructed frame of the next frame, the final frame is kept in the GPU global memory to be used as a reference for the next frames. The final frame memory space is allocated in the *Decoded Picture Buffer*, which is rewritten whenever the final frame is not used as a reference anymore.

Once the final frame data is obtained, it is sent to the CPU for storing or for further processing (see "*GPU⇒CPU*" in Figure 4.1). Naturally, this part can be omitted whenever the GPU is also responsible for displaying the video. In such case, the final decompressed frame can be kept in its global memory and forwarded to the display subsystem. While the memory transfers and the GPU decoding kernels are performed for each frame, the CPU continues its processing, i.e., entropy decoding of the bitstream portion that corresponds to the next frame. As depicted in Figure 4.1, this pipelined procedure is applied for decoding all frames in sequence: while the GPU decodes the current frame (f), the CPU entropy decoder takes care of the next frame ($f+1$).

To fully exploit the GPU computational capabilities in the real-time GHEVC decoding, an additional level of parallelism is considered here, by having different portions of the current frame decompressed simultaneously. When multiple CUDA Streams are applied, the commands corresponding to different streams (kernels and CPU⇔GPU memory transfers) may run concurrently,

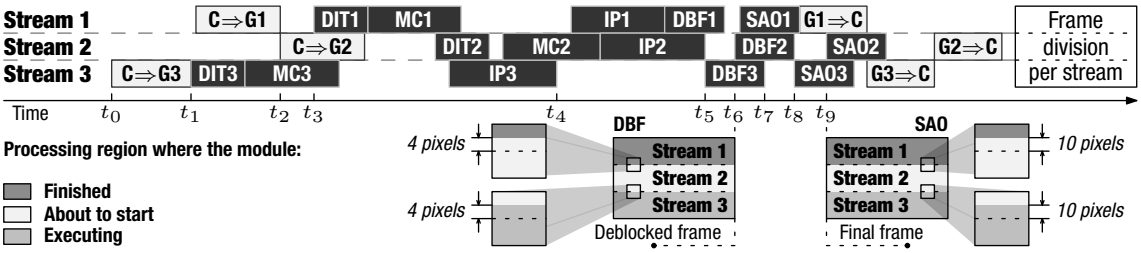


Figure 4.2: Asynchronous CUDA Stream processing in the proposed GHEVC video decoder.

according to the GPU capabilities [36] (see Chapter 2). Hence, to allow the simultaneous and independent processing of each CUDA stream, the frame is divided into sets of CTUs.

Although most GPU decoding modules (i.e., DIT, MC, DBF and SAO) can process any sets of CTUs in parallel, strict data dependencies imposed by the IP module (see Chapter 3) restrict any frame partitioning. In fact, due to the IP module dependencies, the parallel processing within a single frame can only be efficiently exploited in the GPU by this module at the level of CTU rows [28]. Hence, each CUDA stream must be assigned with a set of consecutive CTU rows. By adopting such a pipelined processing scheme, the DIT and MC can be implemented in parallel on different parts of the frame, before the IP is performed. Furthermore, the DBF and the SAO modules can also simultaneously process different portions of the reconstructed frame. In Figure 4.2, an example of a frame processing with three CUDA Streams is presented.

Although the operations within a single Stream are launched *in order* by the CPU, they may be scheduled *out of order* across different streams. This situation is illustrated in Figure 4.2, where the *C⇒G3* memory transfer of *Stream 3* starts before *Stream 1* and *Stream 2* (at time t_0). Between t_1 and t_2, the *DIT3* and part of the *MC3* GPU kernels of *Stream 3* are completely overlapped with the memory transfer of *Stream 1*, which leads to an overall processing time reduction.

Besides the overlapping of CPU⇔GPU memory transfers with GPU kernel executions, even the GPU kernels from different Streams can be overlapped. However, the number of overlapped GPU kernels is limited by the amount of available GPU resources (i.e., the resources that are not busy with the kernels in execution in the GPU). As illustrated in Figure 4.2, although the *DIT1* kernel from *Stream 1* can start at t_2, it only starts at t_3, because the GPU is still busy with the *MC3* kernel of *Stream 3*. In this case, only at t_3 there are enough GPU resources to start the *DIT1* kernel from *Stream 1*. A similar behavior can be observed along the time for the other modules of each stream in Figure 4.2.

In general, the IP kernel of a *Stream* i can not finish before the IP kernel of the *Stream* $i-1$, due to the intrinsic data dependencies among them. Nevertheless, it may happen that the first 8-pixels row of a Stream's CTU set contains only inter predicted blocks. In this case, the IP kernel is independent from the IP kernels of other streams, as presented for the *IP3* kernel in Figure 4.2, which finishes its execution at t_4.

However, when the decoding is simultaneously performed on different portions of a single frame, special attention must be paid to preserve the compliance with the HEVC standard, i.e., the one already attained at the level of individual intra decoding modules. Since the DBF module operates on 8×8 blocks, which are shifted by four pixels in the vertical and horizontal components (as explained in Chapter 3), the processing region per Stream is also shifted up by four pixels. It is important to notice that even for the 4:2:0 chroma subsampling, the chroma processing region is also shifted by four pixels, because the HEVC standard specifies its filtering procedure in the 8×8 grid of the chroma frames as well [50]. This implies that the *DBF*i kernel of *Stream* i has to wait for the processing completion of the reconstructed frame part from *Stream* $i-1$. In Figure 4.2, this effect is observed in *DBF3*, which does not start between t_4 and t_5, until *IP2* has finished. At the bottom of Figure 4.2, the processing region per Stream is also shown (at t_6) for the *Deblocked frame*, where the *DBF2* is about to start, *DBF3* is executing and *DBF1* is already finished. To ensure the correct GPU kernel execution order, explicit synchronization points are set, with the DBF from *Stream* i starting after the IP execution from *Stream* $i-1$.

In the SAO kernel, the processing region is shifted by one pixel, in order to guarantee the correctness of the procedure if the SAO Edge Offset is selected in the border of the processing region. However, in order to ensure the coherency between the luma and chroma processing regions, an overall shift of 5 pixels is applied in the chroma (4 for DBF + 1 for SAO), corresponding to a shift of 10 pixels in luma for the 4:2:0 chroma subsampling. In this way, a similar procedure is performed for the SAO module, where the *SAO*i kernel of *Stream* i waits until the deblocked frame part of *Stream* $i-1$ is available. For example, in Figure 4.2, the *SAO3* kernel is put on hold from t_7 to t_8 until *DBF2* is done, by applying explicit synchronization points between the SAO from *Stream* i and the DBF from *Stream* $i-1$. Moreover, at the bottom of Figure 4.2, the processing regions of the luma component in the *Final frame* are shown at t_9, where *SAO2* is about to start, *SAO3* is executing and *SAO1* has already finished. Here, it is possible to observe how the processing regions have been shifted up over the *Final frame* in comparison with the original *Frame division per stream* (in dashed lines).

Finally, to support the explicit synchronization between streams for the IP, DBF and SAO kernels, CUDA events are used in addition to the *cudaStreamWaitEvent* function. Hence, kernels from one stream can be halted until a certain event reports its completion (in this case, a kernel of another stream).

4.2 De-quantization and Inverse Transform

The work presented in [29] performs the HEVC DIT procedure using OpenCL, by relying on four GPU kernels: a single kernel per each TB size (i.e., from 4×4 to 32×32). The herein proposed parallel GHEVC DIT algorithm relies on a single GPU kernel for all TB sizes, which avoids the

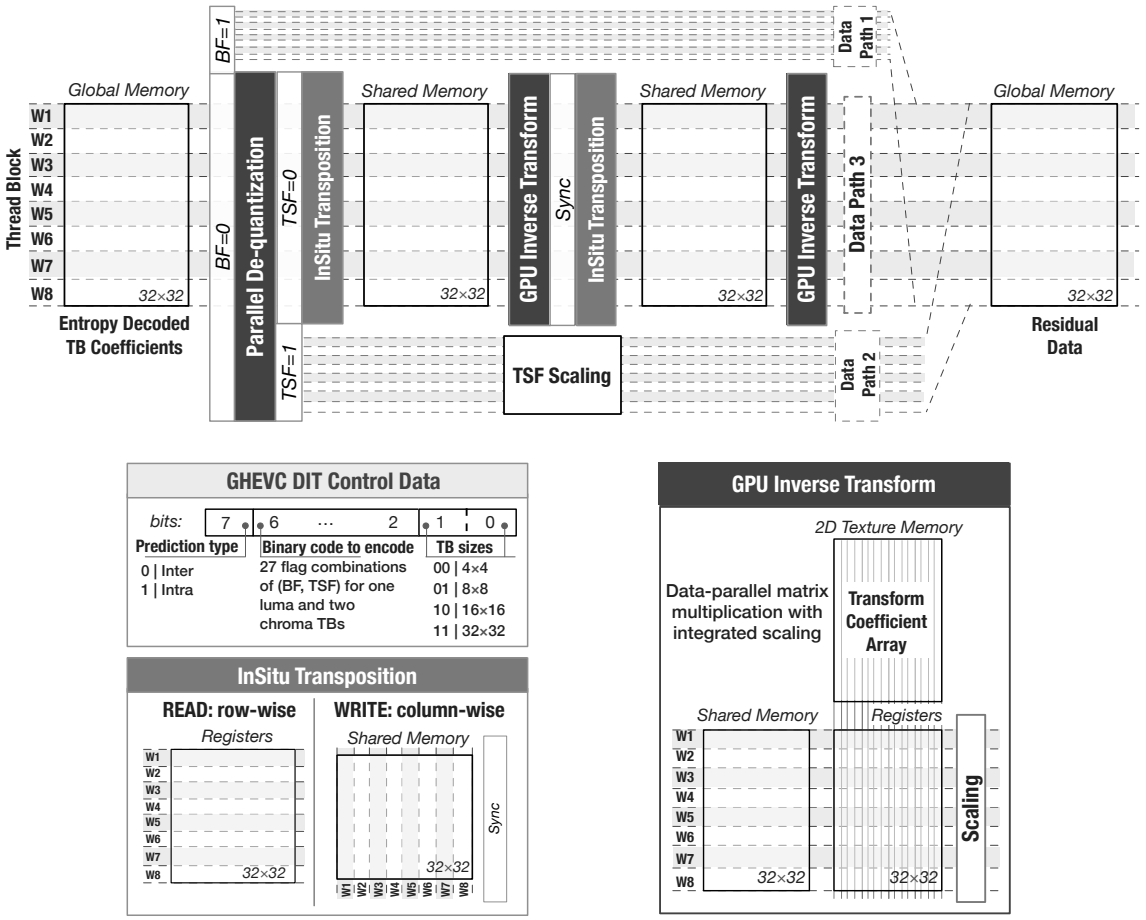


Figure 4.3: Overall de-quantization and 2D inverse transform implementation in the GPU for one 32×32 luma TB.

overhead of launching multiple kernels. The preliminary GPU DIT approaches presented in [28] and [24] do not support 4×4 inverse transforms of inter predicted CUs, since only intra frames (i.e., intra predicted blocks) were considered at these publications. The proposed GPU-based DIT module herein presented is based on [22], which already supports TUs from inter predicted CUs. Moreover, it already has a high degree of fine-grain parallelism since all the TBs in the frame can be processed in parallel.

The general layout of the proposed parallel DIT is presented in Figure 4.3 for the case of a 32×32 TB. In the proposed GHEVC DIT, a single ThB contains 8 warps (Wi), which perform the DIT computations of the TB parts (e.g., eight 32×4 TB sub-blocks in Figure 4.3). The overall procedure starts by asynchronously reading the *Entropy Decoded TB Coefficients* from the GPU global memory. Here, all 32 parallel threads in a warp fetch four rows of its corresponding 32×4 TB part. The parallel DIT output (*Residual Data*) is produced by the different warps, after executing the *Data Paths* 1, 2 or 3 in Figure 4.3.

The considered *Data Path* is selected by the DIT flags, i.e., TBF, CBF and TSF (see Chapter 3). In the proposed implementation, the TBF and CBF of a single TB are merged in a single flag, which

is named Bypass Flag (BF), as it was proposed in [29]. Furthermore, to reduce the communication overheads, a specific 8-bit *GHEVC DIT Control Data* structure was designed, such that all the 27 BF and TSF combinations are integrated. In brief, for each TB, there are 3 possible flag combinations that can occur, i.e., (BF, TSF) $\in \{(1,*);(0,1);(0,0)\}$, giving a total of 27 ($3^3$) combinations for one luma and two chroma TBs. This information is binary-encoded with five bits and stored in bit positions 2–6 of the *GHEVC DIT Control Data* structure. Furthermore, the bit positions 0 and 1 of this structure are reserved to encode the TB size information (see Figure 4.3). The remaining bit position (7) is used to designate the prediction type (i.e., Intra or Inter).

This *GHEVC DIT Control Data* is packed for each 4×4 block of the frame. This data is used by the warps to extract the BF and the TSF values during the GPU kernel execution (i.e., to select the *Data Path* for each 32×4 TB sub-block in Figure 4.3). Whenever the BF is set (*Data Path 1*), the fetched TB coefficients are directly forwarded to the residual data (i.e., TBF$=1$ or CBF$=0$). This execution path is also used to integrate the I_PCM mode, where the I_PCM data is stored as the TB coefficients on the CPU side. When the BF is unset, the *Parallel De-quantization* is performed before the TSF is evaluated (see Figure 4.3). In the *Parallel De-quantization*, each thread in a warp independently de-quantizes one TB coefficient.

If the TSF is set, the warps asynchronously perform the *TSF Scaling* in *Data Path* 2. Otherwise (when TSF=0), *Data Path* 3 is selected, where the parallel 2D GPU inverse transform is applied. In the GHEVC DIT, the HEVC *1D Column* and *1D Row Inverse Transforms* are performed by the same *GPU Inverse Transform* procedure. This is achieved by applying the *InSitu Transposition*, where each warp re-arranges the data to fit the correct form. As presented in Figure 4.3, the first *InSitu Transposition* is applied to the de-quantized TB coefficients (in the GPU registers). Here, each warp re-arranges the coefficients to a column-wise representation in the shared memory. To ensure the correctness of the HEVC *1D Column Inverse Transform*, all warps must finish the *InSitu Transposition* (*Sync*) before the *GPU Inverse Transform*.

In the *GPU Inverse Transform* implementation, the shared memory is always read in a row-wise pattern by each warp. Then, the data-parallel matrix multiplication is performed over the read data and the selected *Transform Coefficient Array* (stored in the GPU texture memory). Afterwards, the HEVC *Intermediate Scaling* (see Chapter 3) is independently applied to the obtained results by each warp.

To proceed with the computation of the HEVC *1D Row Inverse Transform*, all warps must finish the previous *GPU Inverse Transform* (*Sync*) and apply the second *InSitu Transposition* (see Figure 4.3). The *Sync* point guarantees that the first 1D transform is finished in all warps before the results are written back to the shared memory. Then, the second *GPU Inverse Transform* is applied with the integrated HEVC *Final Scaling* (see Chapter 3). Finally, each warp finishes by asynchronously writing the produced residual data in the GPU's global memory.

Figure 4.4 presents an example of distinct warp assignments for different TB sizes (i.e., 4×4,

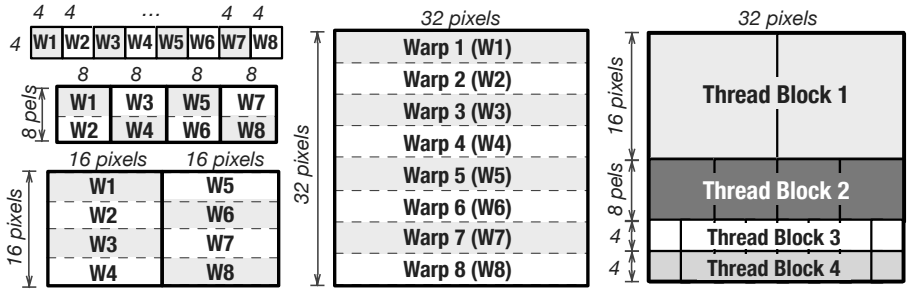


Figure 4.4: Example of GHEVC DIT warp and ThB assignments.

8×8, 16×16 and 32×32). When the GPU kernels start, all eight warps within a ThB start by obtaining the TB sizes stored in the *GHEVC DIT Control Data* (for each 4×4 block of the 32×4 frame segment). According to the obtained TB size, the warps are assigned to different portions of the TB. Figure 4.4 also shows an example of how 4 ThBs can be assigned for a 32×32 partitioned TB.

It is worth emphasizing that the proposed parallel GHEVC DIT algorithm implementation extensively exploits an efficient utilization of the GPU memory hierarchy, by organizing the data accesses as follows: *i*) the *GHEVC DIT Control Data*, the TB coefficients and the residual data are stored in the global memory; *ii*) the transform coefficient arrays are kept in the read-only 2D texture memory; *iii*) the low latency shared memory is used to store the intermediate data for the 2D inverse transform; and *iv*) all remaining data (such as frame size, QP and scaling factors) are stored in the constant memory and broadcast to each warp in one single memory transaction.

4.3 Motion Compensation

The proposed MC approach leverages the fine-grain parallelism of this computationally complex module, while providing fully standard compliant HEVC decoding. To increase the performance, the proposed design maximizes the number of active warps, while ensuring that all threads in a warp perform the same operation from the GPU code (kernel). Furthermore, the data accesses are carefully managed to efficiently exploit the complex GPU memory hierarchy, i.e., global, cache, shared and constant memory.

In [26], the whole *Decoded Picture Buffer* is transferred to the GPU memory at the beginning of the decoding of each frame. Furthermore, the predicted frame has to be sent back to the CPU, in order to perform the remaining modules. In contrast, in the herein presented GHEVC decoder, the frame is entirely decoded in the GPU and it is kept in the GPU memory as long as it is needed as a reference frame. In this way, the proposed decoder allows a significant reduction of the superfluous memory transfers of the reference frames and predicted frame between the CPU and the GPU. Additionally, a complete GPU-based *Decoded Picture Buffer* is provided. Moreover, the proposed MC module performs both the inter prediction and the reconstruction of inter coded CUs. The

required data to generate the reconstructed block are: *i*) the motion data (i.e., motion vectors, reference indexes, reference frames and prediction direction); *ii*) PB partitioning mode; and *iii*) residual data.

Although the prediction of each inter PB can be computed in parallel, the shape and size of smaller PBs are limiting factors for the overall GPU performance, due to the irregularity of the memory accesses to the reference frames. Nevertheless, the warp can process more than one PB and store the prediction values in the GPU shared memory, in order to maximize the utilization of the GPU global memory bandwidth for the subsequent decoding procedures, i.e., reconstruction.

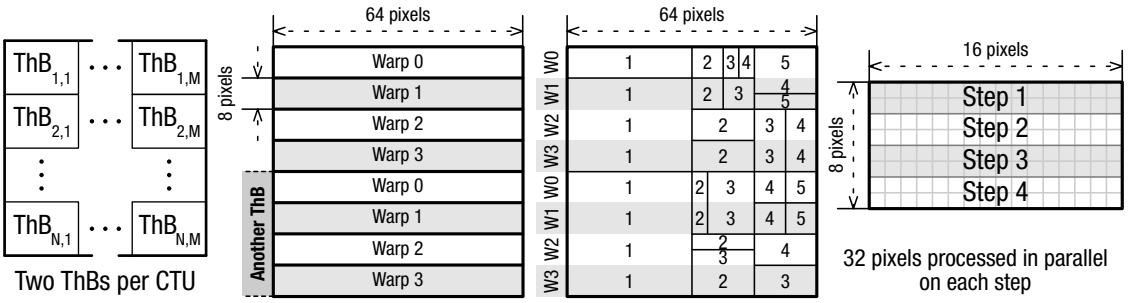
Since the GPU global memory is accessed, at minimum, via 32-byte memory transactions, the block reconstruction procedure is performed with blocks of 32-pixels width. With this approach, it is guaranteed that the residual block row is fetched in a single memory transaction, and that the reconstructed block row is stored with a single memory transaction to the GPU global memory. For the chroma component (with chroma subsampling 4:2:0), the warp operates at a luma block of 64×N, which leads to a 32×N/2 chroma block.

Due to the lower latency of memory accesses, in comparison with the GPU global memory, the GPU shared memory is also used as temporary storage for the interpolation procedure. Here, a pixel block from the reference frame is fetched from the GPU global memory to the shared memory. However, the GPU shared memory has a quite small size, which can reduce the number of simultaneously active warps if a high amount of shared memory per warp is requested by the kernel, i.e., the size of the 64×N block.

In the proposed GPU-based MC module, the best performance is achieved when a warp operates a 64×8 luma block, which is a trade-off between: the parallelism degree, the amount of requested shared memory, the number of active warps and the global memory bandwidth. Therefore, four warps are assigned per ThB, which performs the GPU-based MC module in a 64×32 pixel block of the frame. Furthermore, each warp computes the prediction of each PB or sub-PB, which relies in its 64×8 pixel block, e.g., for a 64×64 PB, eight warps (two ThBs) perform the inter prediction and the reconstruction of each 64×8 sub-part of the PB.

Hence, as it is shown in Figure 4.5, a single ThB composed of four warps is assigned to process each 64×32 luma pixels (see Figures 4.5(a) and 4.5(b)). Each warp predicts a 64×8 pixel luma sub-block and its corresponding chroma sub-blocks. If a $N \times N$ PU is larger than eight pixels in the vertical axis, each warp Wi will perform the prediction of its N×8 sub-blocks (see Figure 4.5(c)). Each pixel in a sub-block is predicted by one thread of the warp, where 32 pixels are predicted in each step. Hence, a 16×8 sub-block is predicted in four steps (see Figure 4.5(d)).

The motion data that is required to perform the proposed inter prediction of a PU is packed into a 64-bit word, as presented in Figure 4.6 (*MC Control Data*). Since the smallest PB partitions are 8×4 and 4×8 pixels in the HEVC inter prediction, two *MC Control Data* words are assigned to each 8×8 luma pixel block of the frame, in order to perform the motion compensation for each



(a) Frame-level parallelism. **(b)** CTU processing. **(c)** Warp-level processing order of a CTU. **(d)** Thread-level processing.

Figure 4.5: Thread blocks/warps assignment and the proposed GPU Motion Compensation functionality.

smaller block, i.e., two 4×8 or two 8×4 blocks (see Section 3.1). Moreover, the *MC Control Data* for the whole frame is stored in the GPU global memory, which can be retrieved in a single memory transaction to perform the motion compensation of a single PU. The 64-bit word of the *MC Control Data* is structured as:

- **Block size** (bits 0 to 4): encodes all possible 24 PU partitions (i.e. 64×64, 64×48, 64×32, 64×16 and so on).
- **Prediction type** (bit 5): signals if the CU is intra (1) or inter predicted (0).
- **Prediction direction** (bits 6 and 7): indicates if List 0 (bit 6) and List 1 (bit 7) reference frames are used.
- **Ref Idx L0** (bits 8 to 11): index of the chosen reference frame from a set of 16 possible values from List 0 (L0).
- **Ref Idx L1** (bits 12 to 15): index of the chosen reference frame from a set of 16 possible values from List 1 (L1).
- **L0 Y**, **L0 X**, **L1 Y** and **L1 X** (bits 16 to 63): store the vertical (Y) and horizontal (X) motion vectors at quarter-pel resolution of List 0 (L0) and List 1 (L1).

Figure 4.6 also presents a simplified flowchart of the overall inter prediction procedure that is adopted by the proposed GPU MC kernel. At the beginning, each warp is assigned to its own 64×8 pixel block of the frame (see *Warp Assignment*). Then, the information required to process one block (*MC Control Data*) is transferred from the GPU global memory (see *Acquire motion information*).

Upon the fetch of the *MC Control Data*, bit 5 is checked to verify if the block belongs to an intra predicted CU. In this case, the overall MC procedure is bypassed (see "*Is Intra?*" decision in Figure 4.6). Otherwise, the block is inter predicted and bit 6 signals if the reference frame belongs

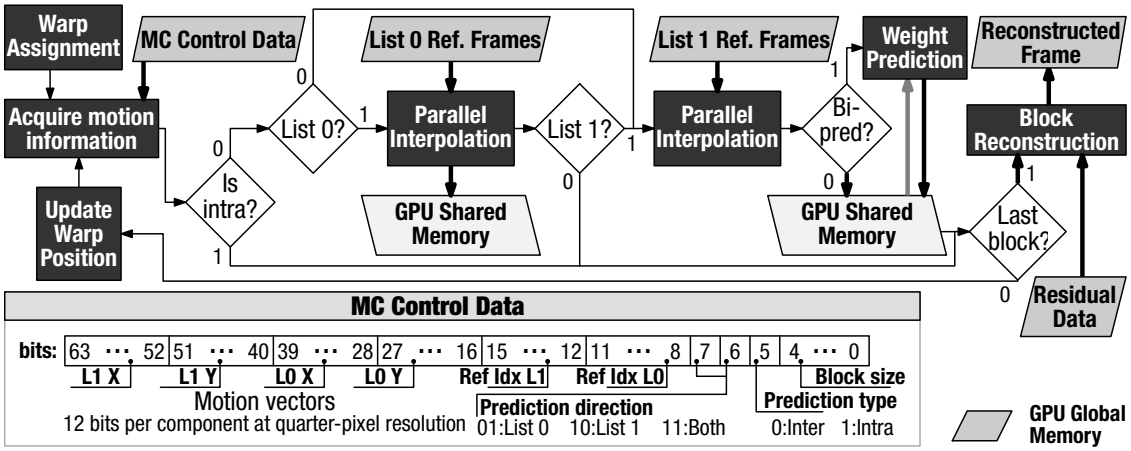


Figure 4.6: Flowchart and control data of the GHEVC MC module.

to List 0 (see "*List 0?*" decision in Figure 4.6). If bit 6 is set, the *Parallel Interpolation* procedure [26] is performed on the selected reference frame from List 0, according to its motion information (*L0 X*, *L0 Y* and *Ref Idx L0*), and the predicted block is stored in the *GPU Shared Memory*.

Later, bit 7 (see "*List 1?*" decision in Figure 4.6) is inspected to decide if the inter prediction of List 1 should be performed. If bit 7 is set, the *Parallel Interpolation* is executed, by applying the motion vectors and the reference frame of List 1 (*L1 X*, *L1 Y* and *Ref Idx L1*). Then, bit 6 (List 0) is verified to check if the bi-prediction has to be performed (see "*Bi-pred?*" decision). If bit 6 is unset, the predicted block is stored in the *GPU Shared Memory*, since the bi-prediction is not performed. Otherwise, if bits 6 and 7 are set, the bi-prediction is implemented by the *Weight Prediction* procedure, i.e., by averaging both predicted blocks from List 1 (in GPU registers) and from List 0 (previously stored in the *GPU Shared Memory*), where the output bi-predicted block is stored in the *GPU Shared Memory*.

Afterwards, the warp checks if all PBs or part of PBs inside its 64×8 pixel block have been predicted (see "*Last block?*" decision). If there are pending blocks to be processed, the overall process is repeated, where the warp position in the frame is updated for the next block (*Update Warp Position* in Figure 4.6).

When all blocks of the 64×8 pixel block that was assigned to the warp are predicted (i.e., "*Last block?*" decision returns one), the reconstructed block is computed by adding the 64×8 predicted blocks from the *GPU Shared Memory* and the 64×8 residual block of *Residual Data* from the GPU global memory (see *Block Reconstruction* in Figure 4.6). At this respect, it is important to note that 32 pixels from luma or chroma components are simultaneously reconstructed and the accesses to both GPU memory spaces (shared and global) are performed with a single memory transaction, to improve the performance.

Moreover, the overall procedure is orchestrated to avoid warp divergence, since all threads in a warp always follow the same execution path in both prediction and reconstruction steps, while

the GPU shared memory accesses were carefully designed in order to avoid bank conflicts [26]. The final reconstructed 64×8 luma pixel block is stored in the GPU global memory, as part of the *Reconstructed Frame* for being used by the subsequent GHEVC decoder modules.

4.4 Intra Prediction

Due to the strict dependencies between the reconstructed blocks, the proposed GHEVC IP design adheres to the wavefront execution paradigm, as it was proposed in [28] and [24]. However, contrasting to the preliminary approaches in [28] and [24], where only intra frames were considered, the GHEVC decoder herein presented already supports intra predicted blocks inside inter frames, as in [22]. This capability is achieved by coupling the IP GPU kernel after the MC GPU kernel execution and by explicitly considering the cases where neighboring blocks are intra or inter predicted. In accordance, the intrinsic IP data dependency checking procedure had to be updated too. Furthermore, the GPU thread assignment of the IP kernel was also improved, in order to ensure a better load balancing across the available SMs.

Just like in the MC module, the proposed IP module performs the frame prediction and the reconstruction. However, while the reconstructed block in the MC module is not reused, it is used in the IP module as an input when predicting the neighboring blocks. Since the size of the TB can only be equal or smaller than the PB in an intra predicted CU, the IP is performed at the TB level, instead of PB [48] (as explained in Chapter 3). Hence, by following the z-scan order, the IP module carries out the prediction for each TB in a PB, for each PB in a CU and for each CU in a CTU. The block size is determined from the data already available in the global memory (i.e., bits 0 and 1 of the *GHEVC DIT Control Data*). This design option arises from the fact that the IP is performed after the DIT.

As it was referred before, to perform the intra prediction, the pixels from the reconstructed neighbor blocks are used as references. These intrinsic data dependencies between intra predicted pixel blocks limit the level of parallelism within the IP module. Nevertheless, a coarse-grain level of parallelism can still be obtained by executing the intra prediction in a wavefront approach for the whole frame [28], instead of processing a single CTU row at a time. Thus, the intra prediction of a given frame pixel block can be executed as soon as the required neighboring blocks are reconstructed, regardless of the remaining blocks in the frame.

In the proposed GPU implementation, a single warp is responsible to perform the intra prediction of any PB or PB part inside a set of N pixel rows of the frame. Since multiple warps can simultaneously process the IP of the next blocks (as soon as the data dependencies are satisfied), it is necessary to keep track of the position of the currently processed block within the frame. This warp "position" value assumes the block enumeration scheme from the left to the right side of the frame, by strictly taking into account the block dependencies [28]. This approach provides two

main advantages:

1. The data dependencies for the current block are checked by verifying the “positions” of the neighboring warps in the frame, which means that all pixel blocks with a lower value than the warp “position” are already reconstructed.

2. The GPU cache pollution is reduced, since only the dependencies of the pixel blocks in the warp “position” of the frame are checked.

In particular, it was decided to process N=8 pixels in each row, since it provides the best trade-of between the granularity of the wavefront processing (parallelism degree) and the amount of GPU global memory accesses to check dependencies. When compared to [24], one of the contributions of the herein proposed IP kernel is a better load balancing across the SMs. A neighborhood of 8 pixel rows are processed by different thread blocks. In this case, the workload of the wavefront approach is efficiently distributed according to the existing GPU resources.

Since the intra prediction is performed at a TB level, the necessary data to execute the IP module are: *i*) the TB size and prediction type, which are provided by the *DIT Control Data*; *ii*) the intra prediction mode; and *iii*) the reconstructed frame. In Figure 4.7(a), the luma *IP Control Data* is presented, organized in a single byte word, as follows:

- **Prediction mode** (bits 0 to 5): encodes 35 intra prediction modes (Planar, DC and 33 Angular modes) and an extra mode for the PCM.

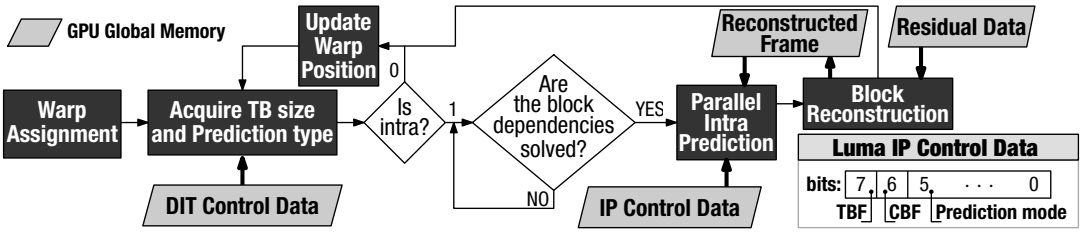
- **CBF** and **TBF** (bit 6 and 7): reserved for further use in the DBF and SAO modules.

Since the smallest luma PB size is 4×4, the luma *IP Control Data* is stored in the GPU global memory as an 1-byte word per each 4×4 pixel block of the frame, while the chroma *IP Control Data* is stored for an 8×8 pixel block of the frame, when considering the 4:2:0 chroma subsampling format.

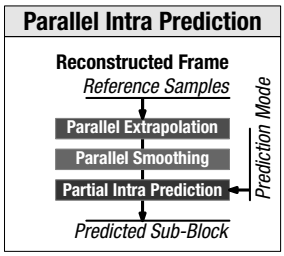
In order to support inter prediction blocks and to better distribute the load between the SMs, the GPU global memory is used to gather the information regarding the “position” of all warps, which allows dependency checks between warps from different ThBs. In contrast, the implementations presented in [28] and [24] perform the dependency checks in the GPU shared memory between warps inside the same ThB and in the GPU global memory between warps across ThBs. Furthermore, the memory coherency for these dependency checks is ensured by using the *volatile* keyword and CUDA memory fence functions [36].

The flowchart of the IP module is presented in Figure 4.7(a), where 8-pixel rows are assigned to each warp by the *Warp Assignment*. Starting from the first pixel block (on the left of the 8 pixel row) until the end of the pixel row, the procedure described in the following paragraphs is applied.

The *DIT Control Data* is fetched from the GPU global memory with a single memory transaction (see *Acquire TB size and Prediction type* in Figure 4.7(a)) to obtain the *Prediction type* (bit 7) and



(a) Flowchart and control data of the GHEVC IP module.



(b) Parallel intra prediction.

Figure 4.7: Framework of the GHEVC IP module.

TB size (bits 0 and 1). Bit 7 is checked in the "*Is Intra?*" decision (see Figure 4.7(a)), in order to verify if the TB belongs to an intra or inter predicted CU. If the TU is part of an inter predicted CU (bit 7 is unset), the warp updates its "position" to the next TB (*Update Warp Position*) and the procedure is repeated by fetching the *DIT Control Data* for the next TB (see Figure 4.7(a)). Otherwise (bit 7 is set), the intra prediction and the reconstruction procedures are performed for the selected TB.

The *TB size* (bits 0 and 1) is used to address the threads inside the pixel block and to determine the required neighboring blocks. Then, the warp "positions" of the corresponding neighboring blocks are verified, in order to check if the dependencies are solved (see Figure 4.7(a)). When the dependencies are satisfied, the reference samples from the neighboring blocks are stored in the GPU shared memory from the *Reconstructed Frame* for faster access. Moreover, the *IP Control Data* is fetched from the GPU global memory, to specify which intra prediction mode will be performed. Then, the *Parallel Intra Prediction* procedure (see Figure 4.7(a)) is performed, with each thread responsible for one or more pixels of the block, as proposed in [28].

As presented in the flow-graph of Figure 4.7(b), each warp will proceed with the *Parallel Intra Prediction* only when all data dependencies are satisfied, i.e., the $4N+1$ reference samples are produced. After fetching the *Reference Samples*, the *Parallel Extrapolation* procedure is performed to fill the whole $4N+1$ reference sample set. At this stage, the nearest available reference sample is broadcasted to all threads in a warp, and each thread simultaneously copies this value to a single unavailable reference sample position (see Section 3.1.4). Then, the *Parallel Smoothing* is performed in the luma blocks, being each thread responsible for applying the HEVC smooth filtering

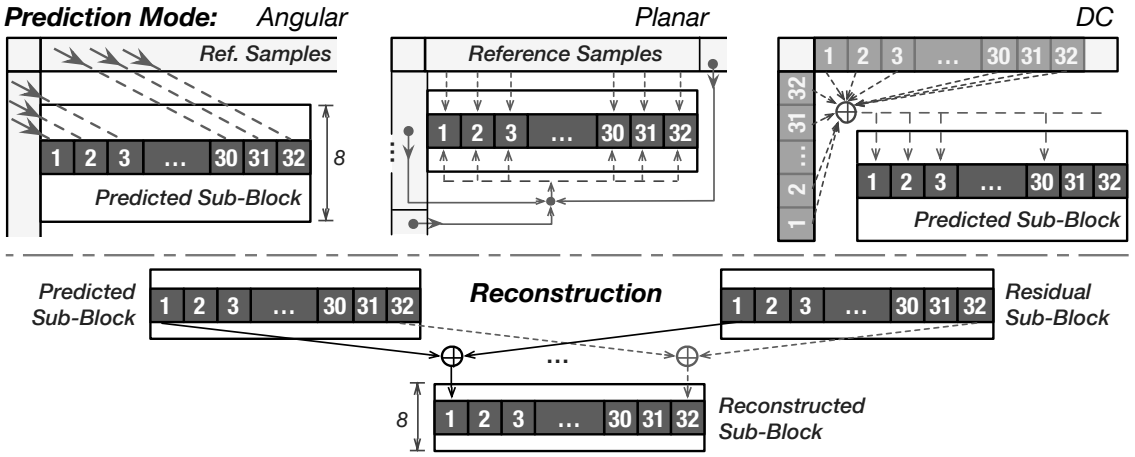


Figure 4.8: Proposed GHEVC Partial Intra Prediction design for the *Mode Prediction* and *Reconstruction* steps.

to a single reference sample. To avoid the need for synchronization points and excessive register usage, each warp has a separate array of reference samples stored in the shared memory. Then, the *Partial Intra Prediction* is performed on each N×8 sub-block, to produce the corresponding reconstructed sub-block.

The *Partial Intra Prediction* is performed in two steps: *Mode Prediction* and *Reconstruction*. In the *Mode Prediction* step, the predicted sub-block is produced by performing one of the three possible modes (Angular, DC or Planar). The mode is selected according to the corresponding luma or chroma prediction mode, which are stored in a 2-byte word for each 4×4 block. In each byte, 6 bits are required to encode 36 intra modes (including I_PCM), while the remaining bits are used to store the CTU TBF flag. The TBF flag is also added to this structure, since both the considered intra modes and the TBF flag are required to execute the DBF and SAO modules. As a result, this 2-byte structure is also re-used in the proposed DBF and SAO parallel algorithms.

As shown in Figure 4.8 for a 32×8 sub-block, the execution of the identified selected mode is done in parallel, being each thread in a warp responsible for one pixel. In the *Angular Mode*, each thread interpolates the reference samples according to the given direction. In the *Planar Mode*, each thread applies a bilinear model according to the relative spatial position of the pixel to be predicted and the reference pixel samples. In the *DC Mode*, all threads cooperatively compute the reference pixel samples average to produce the predicted pixels. In the second step, the *Reconstruction* is also performed in parallel, by adding the predicted sub-block with the corresponding residual sub-block. The obtained reconstructed sub-block is stored in the GPU global memory.

Finally, the warp "position" in the 8 pixel row of the frame is updated to the next TB by the *Update Warp Position* procedure and the overall process is repeated until the warp "position" reaches the end of the 8 pixel row (see Figure 4.7(a)). As mentioned before, when the I_PCM mode

is active, the residual data is marked as the reconstructed sub-block and the *Mode Prediction* is bypassed.

4.5 Deblocking Filter

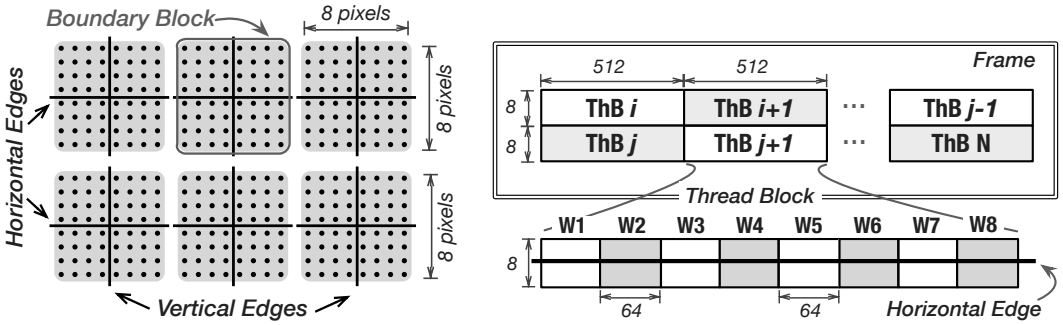
As it was referred in Chapter 3, the DBF module considers up to four samples within a 4×8 (or 8×4) pixel region, in order to filter up to three samples on each side of the boundary (see Figure 3.12). According to the HEVC standard [5], this procedure is first applied on all vertical edges in a frame, followed by the horizontal ones (see Section 3.1.5).

Although this processing scheme fits to conventional CPU architectures, it might not deliver enough degree of fine-grained parallelism for exploiting GPU resources. First, inevitable synchronization between the two DBF stages may significantly degrade the overall GPU performance, since it is required either to launch separate GPU kernels or to synchronize the execution over the global memory. Second, this approach involves many data transfers and it does not allow efficient utilization of the GPU memory hierarchy. For example, when the vertical edges are filtered, data needs to be stored in the global memory, and again retrieved for filtering the horizontal edges. Additionally, the memory access pattern for the vertical filtering involves column-wise strided data accesses, which requires several global memory transactions to fetch a single portion of horizontally filtered data.

In [30], the filtering decisions are calculated in the CPU side and subsequently sent to the GPU. In [24], the boundary filtering strength is always set to two, since all blocks are intra predicted. In contrast, in the herein proposed DBF the boundary strength is directly calculated in the GPU. No additional data needs to be received from the CPU, since all required input data (from the other modules) is already present in the GPU memory, as in [22]. Furthermore, the proposed DBF kernel provides full support for both inter and intra predicted blocks (while [24] only supports intra blocks).

The benefits of the considered memory utilization in the proposed GHEVC decoder can be highlighted by the availability of the input data when performing the DBF module. In particular, all the data that is required for the evaluation of the BS value is already available in the GPU global memory, since it is used when performing the previous modules (i.e., DIT, MC and IP). The BS values are calculated by checking:

- **TU or PU boundary**: the TU and PU sizes are acquired from the *DIT Control Data* and *MC Control Data*, which indicate the TU or PU boundary dimensions according to the edge position in the frame.
- **Intra prediction**: the *Prediction type* (bit 7) of the *DIT Control Data* of both boundary blocks are checked.
- **Non-zero coefficients and a TU edge**: the luma CBF is obtained from bit 6 of the *IP Control Data*.



(a) Edge-level parallelism. **(b)** Thread block processing.

Figure 4.9: Thread blocks/warps assignment of the GHEVC Deblocking Filter.

- **Motion discrepancy**: the *MC Control Data* of both boundary blocks are used to check if they have different reference frames, different number of motion vectors or the absolute differences between the motion vector components is greater than one pixel.

Moreover, the TBF and the PCM mode are obtained from the *IP Control Data*. If TBF is set or the PCM is used as prediction and the *pcm_loop_filter_disabled_flag* is set, the pixel samples that belong to those blocks are not filtered.

The proposed DBF design provides small independent regions of the frame for efficient GPU parallelization, by relying on a different approach for the execution of the DBF stages. As presented in Figure 4.9(a) (see *Edge-level parallelism*), when two consecutive horizontal 4×8 filtering regions are considered, one can identify several non-ovelapping blocks that can be filtered in parallel, as proposed in [30]. These 8×8 pixel blocks, herein referred to as Boundary Blocks (BBs), allow performing both horizontal and vertical filtering on a small subset of locally stored and independent input data. Hence, all 8×8 BBs in a frame can be simultaneously processed without any synchronization points and by efficiently using the GPU memory hierarchy.

As it is shown in Figure 4.9(b) (*Thread block processing*), in the proposed DBF GPU design, each ThB is assigned to perform the deblocking filter on a row of 64 BBs (i.e., 512×8 luma pixels). Inside each individual ThB, eight warps are in charge of carrying out the deblocking filter in a row of 8 BBs (i.e., a block of 64×8 luma pixels). In order to cope with the increased warp-level data requirements, the shared memory is used as an additional memory space to store the GPU register values. As a result, each warp has its own 64×8 memory space for storing the intermediate filtered samples.

A general diagram of the proposed DBF module is presented in Figure 4.10. First, the warps are assigned to distinct 64×8 regions of the frame by the *Warp Assignment* procedure. Then, the assigned 64×8 pixel block of the *Reconstructed Frame* is fetched from the GPU global memory to the *GPU Shared Memory*, to be processed with subsequently faster accesses in the filtering procedure (see *Fetch 64×8 pixel block* in Figure 4.10). The GPU shared memory is used to

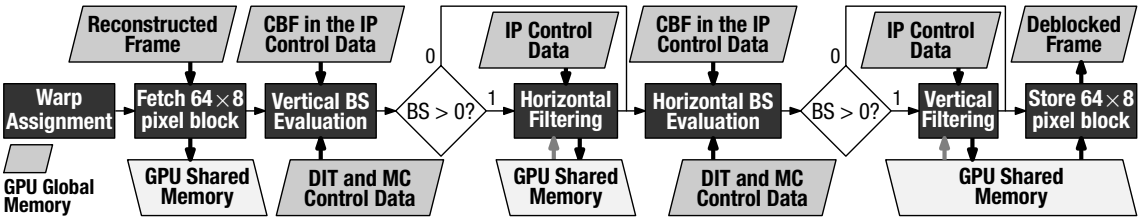


Figure 4.10: Flowchart of the GHEVC DBF module (luma component).

store temporary values during the filtering, where the whole 64×8 pixel block is fetched from the reconstructed frame, filtered and stored back in the GPU global memory as part of the deblocked frame.

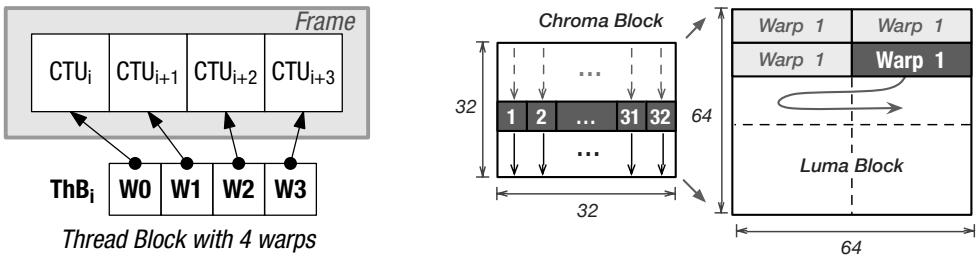
In the proposed algorithm, the whole shared memory that is assigned to a warp is used for both luma and chroma components. Hence, a single 64×8 space is used to perform the DBF for the luma component. On the other hand, the same amount of space can be equally divided for the two chroma components in 4:2:0 subsampling, i.e., the 32×8 chroma U and the 32×8 chroma V blocks are retrieved in parallel and the DBF procedure is simultaneously applied to both chroma components.

All the BS values of each vertical edge in the 64×8 pixel block are simultaneously evaluated (see *Vertical BS Evaluation* in Figure 4.10), where *IP*, *DIT* and *MC Control Data* are obtained from the GPU global memory. If the BS value is greater than 0, the *Horizontal Filtering* procedure is performed on the data stored in the GPU shared memory, as in [24]. After the *Horizontal Filtering* or if the BS is equal to zero for a vertical boundary, the *Horizontal BS Evaluation* is performed for the horizontal edges (see Figure 4.10). Similarly to the *Vertical BS Evaluation*, the BS values of all horizontal edges are calculated according to *IP*, *DIT* and *MC Control Data* (see *Horizontal BS Evaluation* in Figure 4.10). If the BS value is greater than zero, the *Vertical Filtering* is executed [24]. Finally, the filtered 64×8 pixel block is stored in the GPU global memory, as part of the *Deblocked Frame*.

For samples predicted with the I_PCM mode, the DBF is disabled for luma and chroma components according to the *pcm_loop_filter_disabled_flag*, which is stored in the GPU constant memory. Furthermore, the DBF is also bypassed for samples from a lossless-encoded CTU, where the TBF is equal to one. In order to reduce the number of accesses to the global memory, the TBF and the intra modes are packed in the same byte word (*IP Control Data*) and decoded by the GPU with bitwise operations.

4.6 Sample Adaptive Offset

In the proposed parallel algorithm of the SAO module, each ThB (composed by four warps) is responsible for performing the SAO procedure for 4 CTUs in a single row, where each warp is



(a) Frame-level processing.

(b) Warp-level processing.

Figure 4.11: Thread blocks/warps assignment of the proposed GHEVC SAO filtering.

assigned to one CTU, as depicted in Figure 4.11(a) (*Frame-level processing*). In this case, each warp carries out the SAO procedure for 32 pixels in parallel, i.e., one CTU line at a time. Hence, the 32 pixels of each line of the 32×32 chroma CTU are simultaneously processed, as shown in Figure 4.11(b) (*Warp-level processing*). On the other hand, when processing each row of each 64×64 luma block, the SAO filter is first performed on the left-most set of 32 pixels, and then on the right-most set of 32 pixels within the same row.

In order to handle the computational complexity of the SAO procedure and to efficiently use the GPU memory hierarchy, a specific data structure was defined, denoted by *SAO Control Data*, as depicted in Figure 4.12. For each frame component, the proposed 4-byte data structure allows storing the maximum size of each SAO parameter, as specified by the HEVC standard. Hence, the *SAO Control Data* for all luma and chroma component are packed into a 12 bytes word per CTU, which are stored in the GPU global memory. As it is shown in Figure 4.12, the SAO parameters are divided into *SAO Control* and *Offset Data* fields. The chosen data structure of the *SAO Control Data* is presented in Figure 4.12 and it comprises the following fields:

- **Type** (bit 0 and 1): indicates if it is filtered as Edge Offset (*Type*=2), Band Offset (*Type*=1) or neither (*Type*=0).
- **Band**/**Class** (bits 2 to 7): signals which class is used for the Edge Offset filtering, or which initial band is used for the Band Offset filtering [24].
- **Offsets** (bits 8 to 31): stores the four offset values used for the SAO filtering, where 6 bits are used for each offset separated by their sign and absolute value.

The overall procedure implemented for the SAO module (luma or chroma component) is presented in Figure 4.12. After the *Warp Assignment* procedure, the *SAO Control Data* is fetched from the GPU global memory and the respective *Type* is used to select which type of filtering is applied (see *Fetch SAO Type* in Figure 4.12). Accordingly, if *Type* is equal to 2, the *Edge Offset Filtering* is performed on the *Deblocked Frame*, where each pixel is processed by a single thread. The class of the Edge Offset (horizontal, vertical, 135° diagonal or 45° diagonal), as well as its

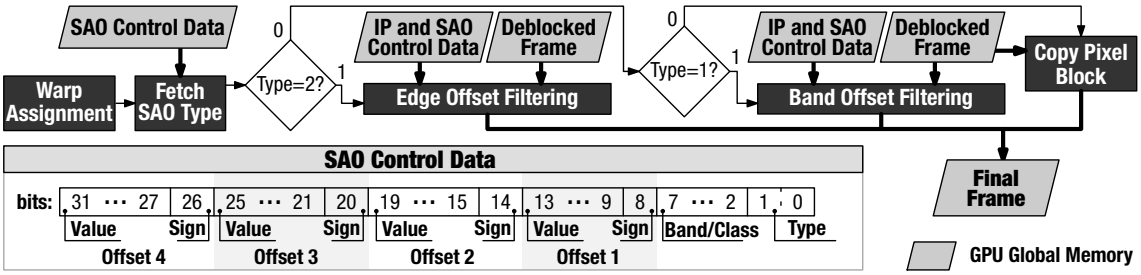


Figure 4.12: Flowchart and control data of the GHEVC SAO module (luma or chroma component).

offset values are obtained from bits 2 to 31 in the *SAO Control Data*.

If *Type* is equal to 1, the *Band Offset Filtering* is executed, where bits 2 to 7 indicate the first band of pixel values to be filtered. Then, the four offsets are added to the pixels whose values belong to one of the four consecutive bands. After the execution of these filtering procedures, the filtered part of the frame is stored in the GPU global memory (see *Final Frame* in Figure 4.12). Finally, if *Type* is equal to 0, the corresponding part of the *Deblocked Frame* is directly copied to the *Final Frame* by the *Copy Pixel Block* procedure.

Similarly to the proposed GPU DBF algorithm, the proposed SAO parallel approach also fetches the *IP Control Data* corresponding to each 4×4 block in a CTU to avoid filtering pixel samples predicted with I_PCM mode, when the *pcm_loop_filter_disabled_flag* is set. Moreover, the lossless mode (TBF=1) is already integrated in the SAO parameters, by setting the *SAO Type* equal to zero.

4.7 Summary

In this chapter, the design of the proposed GHEVC decoder was presented, by leveraging the fine-grained parallelism of the HEVC computationally complex and highly data dependent procedures, while providing full compliance with the HEVC decoding.

The CPU and GPU collaboration was thoroughly explained as a sequence-level parallelism, implemented by means of a pipeline topology. Hence, while the GPU is processing a frame, the CPU is performing the entropy decoding of the next frame. Moreover, frame-level parallelism is extensively achieved with multiple CUDA streams, with different parts of the frame being concurrently processed.

All HEVC decoding modules (except the entropy decoder) of the decoding procedure have been efficiently offloaded to the GPU device. To fully exploit the GPU capabilities, the commonly used CUDA [36] programming model was employed to develop the proposed GHEVC decoder modules, in particular: DIT, MC, IP, DBF and SAO. In order to maximize the GPU performance for each HEVC module, three main features and requirements were specifically considered [36]:

1. **Fine-grain Parallelism**: each HEVC module was implemented in a way that it exposes as

much data parallelism as possible, which allows a large number of simultaneously active threads. This is fundamental to take advantage of the GPU architectures and achieve good performance.

2. **Memory Optimizations**: all the data accesses that are performed by each module were carefully managed, in order to efficiently take advantage of the complex GPU memory hierarchy, i.e., global, cache, shared, register, constant and texture memories. Moreover, memory access latency, coalesced accesses, bank conflicts, register spilling and memory bandwidth utilization were also aspects that have been taken into account.

3. **Instruction Throughput**: the whole GPU programming was conducted by also focusing on reducing the branch divergence, since the different execution paths have to be serialized, thus decreasing the overall performance of the GPU.

The proposed GHEVC procedures that were presented herein are deeply analyzed in the next chapter. Moreover, the GHEVC decoder high performance is compared with available state-of-the-art HEVC decoders.

5

Experimental Evaluation

Contents

To evaluate the performance of the developed GHEVC decoder, the JCT-VC recommended test conditions and configurations [134] were adopted, by considering the setup summarized in Table 5.1. It was considered the HEVC Main profile, which can handle 8-bit depth pixel values sampled with the 4:2:0 chroma subsampling format. From the recommended test video sequence set [134], the sequences with the highest frame resolution were adopted, which includes *Class A* (2560×1600) and *Class B* (1920×1080) resolutions, since they are the most computationally demanding. To further challenge the proposed GHEVC algorithms, a new set of video sequences (*Class S*) was also defined, in order to also include the Ultra HD 4K (3840×2160) frame resolution, obtained from the Sveriges Television AB (SVT) High Definition Multi Format Test Set [135].

All frames of the selected video sequences were encoded with three different configurations: *i*) *All Intra*, only intra frames; *ii*) *Random Access*, a pyramidal structure with I and B frames; and *iii*) *Low Delay*, only the first frame is an intra frame, while the remaining are B frames. Although the *Low Delay* configuration is not recommended for *Class A* resolutions, it was included for all tested sequences for validation and analysis purposes. Furthermore, the several considered video sequences were encoded by setting the QP value from 22 to 37 (see Table 5.1).

To encode these input video sequences, the HM 15.0 reference software [136] was used according to [134], without any Tiles and WPP features, in order to simulate the worst case scenario. The resulting bitstreams were then used in the decoding procedure, to evaluate all the GHEVC modules. Finally, the conceived integration was aggregated to the HM 15.0 decoder in order to evaluate:

1. **Kernel-Level Thread Block Configuration**: to determine the best thread block configuration for each proposed module;
2. **GHEVC Profiling Analysis**: to show the contribution of each proposed module to the overall processing time, when only one CUDA stream is employed;
3. **CUDA Streams Scalability**: to evaluate the achieved performance, by overlapping GPU kernels and memory transfers (CPU⇔GPU) with multiple streams;
4. **Comparison with Previous Intra GHEVC**: to demonstrate the performance improvement over previous implementations;

Table 5.1: Selected setup and video sequences.

HEVC Profile	Main (8-bit depth with 4:2:0 chroma subsampling)
Video Class	S (Ultra HD 4K), A (WQXGA) and B (Full HD)
Class S [135] (500 frames)	CrowdRun, ParkJoy, DucksTakeOff, IntoToTree and OldTownCross
Configuration	All Intra, Random Access and Low Delay
QP	22, 27, 32, 37

Table 5.2: Available NVIDIA GPU devices from Maxwell and Kepler architectures.

Architecture	Name (Compute Capability)	Short Name	Cores (SMs)	Clock	Bandwidth	L2 Cache	Year
Maxwell	GeForce GTX TITAN X (5.2)	Titan	3072 (24)	1000 MHz	336.5 GBps	3.14 MB	2015
	GeForce GTX 980 (5.2)	G980	2048 (16)	1177 MHz	224.0 GBps	2.10 MB	2015
	GeForce GTX 960 (5.2)	G960	1024 (08)	1215 MHz	112.0 GBps	1.05 MB	2015
Kepler	Tesla K40c (3.5)	K40c	2880 (15)	745 MHz	288.0 GBps	1.57 MB	2013
	GeForce GTX 780 Ti (3.5)	G780	2880 (15)	980 MHz	336.0 GBps	1.57 MB	2013
	GeForce GTX 680 (3.0)	G680	1536 (08)	1006 MHz	192.0 GBps	0.52 MB	2012

5. **HEVC Decoding Performance**: to evaluate the best performance obtained with the selected hardware.

Accordingly, the HEVC de-quantization, inverse transform, motion compensation, intra prediction, deblocking filter and sample adaptive offset are completely handled by the proposed GPU parallel modules. The presented evaluation considers the whole decoding structure except the entropy decoder, which was kept at the CPU side because of its highly irregular execution pattern and unsuitability to be efficiently executed at the GPU accelerator. In fact, since the entropy decoder corresponds to the first module of the decoding pipeline, representing less than half of the overall decoding time [10, 16, 19], it was decided to execute it in pipeline with the the remaining decoding modules, i.e., frame $(f + 1)$ was entropy decoded at the same time as the previous frame (f) was processed by the remaining decoding structure (see Chapter 4).

In what concerns the hardware platforms that were used in this experimental evaluation, six different computing setups were adopted by using GPUs from two NVIDIA architectures (i.e., Maxwell and Kepler) and an Intel® Core™ i7-6700K CPU @ 4.00GHz with four cores. To fully exploit the targeted NVIDIA GPU architectures, the proposed algorithms were implemented with CUDA programming model version 7.5 [36]. The six considered GPU devices are presented in Table 5.2, which represent a rather representative range from low-end to high performance GPUs. Finally, the obtained results are presented for each configuration, QP and frame resolution (class), where the obtained performance in a given class of sequences represents the computed average for all tested video sequences with the same resolution.

5.1 Kernel-Level Thread Block Configuration

In this section, the several proposed GPU kernels were evaluated by considering different ThB configurations. The only exception was the DIT kernel, because the number of warps can not be changed without explicitly changing the proposed algorithm. In fact, as it was explained in Chapter 4, the warps of the DIT kernel are assigned according to the size of the TB and they jointly execute the inverse transform by using the GPU shared memory and synchronization points.

Table 5.3: GPU kernel execution time (in ms/frame) when varying the number of warps in a ThB.

Number of warps per ThB	Kernel execution time [ms/frame]			
	MC	IP	DBF	SAO
01	3.31	16.03	0.62	0.52
02	2.48	16.12	0.74	0.77
03	–	15.82	0.60	0.89
04	**2.26**	15.87	0.54	**0.41**
05	–	15.92	0.54	0.41
06	–	15.81	0.53	0.42
07	–	15.96	0.53	0.45
08	2.37	**15.70**	**0.51**	0.44
09	–	16.01	0.52	0.47
10	–	16.02	0.58	0.42

In this case, a higher number of warps would force synchronization between the different TBs, that are asynchronously performed in the herein proposed DIT, while a smaller number of warps would force changes in the algorithm (with also some inherent loss of performance). However, this restrictions is not applicable to the remaining GPU kernels, which are thus considered in the following analysis.

Due to the huge amount of memory accesses in the MC kernel, the reference frame block position calculations heavily exploit bitwise operations, in order to avoid integer divisions and multiplications. For this reason, the MC kernel requires a number of warps that correspond to a power of 2. Finally, the maximum number of warps per thread block is device-dependent for the IP and the MC kernels, due to the GPU resource demands within each warp. Hence, the maximum number of warps obtained in the Titan GPU for the IP kernel and for the MC kernel is 10 and 8, respectively.

The average execution time that is spent by each GPU kernel is presented in Table 5.3 when considering the following configuration:

1. one CUDA Stream;
2. QP value equal to 27;
3. all video sequences from *Class S* (3840×2160);
4. *All Intra* configuration for the IP kernel and *Random Access* configuration for the remaining kernels.

Moreover, the number of registers per kernel was kept fixed, while the amount of shared memory is a function of the number of warps per thread block. Although the difference between the maximum and the minimum obtained performance is less than 1 ms/frame, the number of warps per kernel

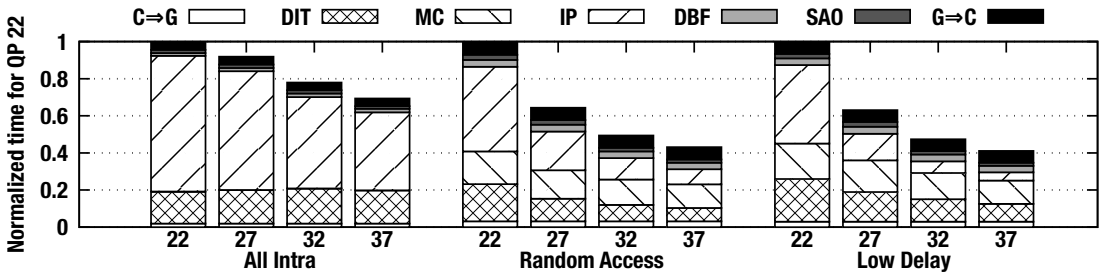
that provide the lowest time is: *i*) 4 warps per ThB for the MC kernel; *ii*) 8 warps per ThB for the IP kernel; *iii*) 8 warps per ThB for the DBF kernel; and *iv*) 4 warps per ThB for the SAO kernel. All these results confirm the considerations and comments previously addressed in Chapter 4.

5.2 GHEVC Profiling Analysis

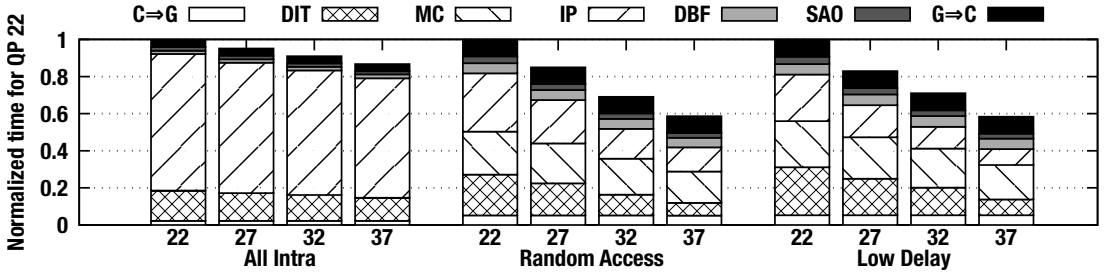
The evaluation and characterization of the performance of each individual module was conducted in a preliminary profiling analysis, by running a single CUDA Stream in the state-of-the-art NVIDIA Titan GPU. The obtained profiling results are presented in Figure 5.1, by using a normalized scale to represent the individual processing time for the different modules (including memory transfers), over the overall frame processing time corresponding to the QP 22 configuration, which is the most time consuming setup. At this respect, it is worth noting that athough the normalized memory transfers time, to and from the GPU ($C \Rightarrow G$ and $G \Rightarrow C$), increases with the QP value for all classes and configurations, this overhead is always constant within a class. This is mainly because the data to be sent to the GPU (as well as the amount of data to be transferred from the GPU, which corresponds to the decompressed frames), does not significantly depend on the QP value. In fact, the amount of input/output data mainly depends on the frame resolution, which means the amount of data and not the actual content. On average, those transfer times correspond to a total of 1.41 ms, 0.76 ms and 0.45 ms per frame for *Class S*, *Class A* and *Class B*, respectively. Nevertheless, in a multiple CUDA streams scenario, most of memory transfers are overlapped with the kernel executions.

As it was expected, the IP module is the most time consuming module in the *All Intra* configuration, which can be observed for *Class S*, *Class A* and *Class B* in Figure 5.1. However, for all tested classes, the IP processing time is reduced with the increase of the QP value. For higher QP values, the HEVC encoder prioritizes the frame rate over distortion, which leads to the selection of greater block sizes per CTU. On the decoder side, the GPU IP module can take advantage of these larger blocks, with more coalesced memory accesses and less dependencies to check. This effect can be better observed for frames with higher resolutions (e.g. *Class S*), as a result of the increased parallelism obtained with a larger wavefront.

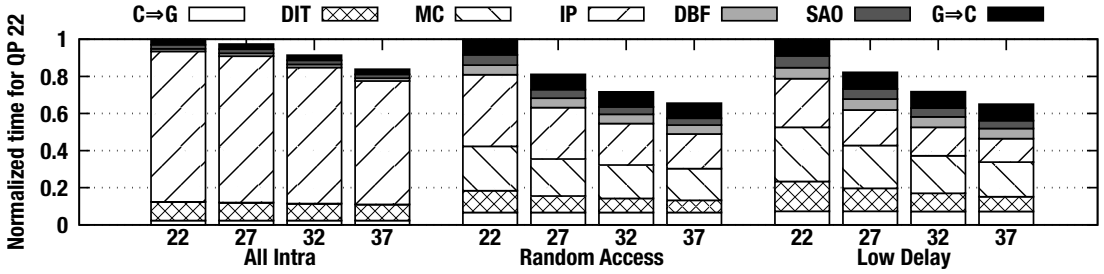
Moreover, it is also observed that the processing time of the remaining modules in the *All Intra* configuration varies slightly with the QP values, on account of the obtained parallelism level, despites requiring significantly lower processing time when compared with the IP kernel. Among these modules, the higher computational demands of the DIT module result in higher processing times, when compared to both the DBF and SAO modules. Even though higher QP values imply larger TB sizes and, consequently, a smaller DIT processing time, this kernel is mainly controlled by the amount of “bypassed” TBs. In fact, due to the limited prediction efficiency exploited by the *All Intra* configuration, a great amount of residual data is obtained, being the TBs rarely encoded



(a) Class S – 3840×2160.



(b) Class A – 2560×1600.



(c) Class B – 1920×1080.

Figure 5.1: Normalized frame processing time, considering the setup with QP=22 as the reference, for *All Intra*, *Random Access* and *Low Delay* configurations of a) Class S, b) Class A and c) Class B.

as “skipped” or “bypassed”. Different results are observed for the *Random Access* and *Low Delay* configurations, given that the inter prediction can provide smaller residual data and more “bypassed” TBs.

In what concerns the *Random Access* and *Low Delay* configurations, both presented a similar behavior in all classes. In those cases, the IP module is also the most time consuming when considering lower QP values. Nevertheless, the IP processing time decreases when the QP value increases, due to the encoder algorithm tendency to exploit inter prediction rather than intra prediction in high QP values scenarios for bitrate saving purposes (see Figure 5.1). Furthermore, when compared with the *Random Access* configuration, the normalized IP processing time is even lower for the *Low Delay* configuration, since it has less intra predicted CUs and only one intra frame.

For all classes in the *Random Access* and *Low Delay* configurations, the processing time of the MC module marginally decreases with the increase of the QP values. In this case, the overall processing time is also reduced for higher QP values, due to the larger PB sizes. However, this reduction is diminished because of the increased amount of inter predicted PBs, which were intra predicted for lower QP values.

In what concerns the in-loop filters (i.e. DBF and SAO), it was observed that the overall processing time is almost constant over the tested QP values, for all configurations and classes. This is mainly due to the fact that the DBF and SAO kernels execution times are strongly constrained by the GPU memory accesses. In this case, the overall performance is basically dominated by the frame resolution.

5.3 CUDA Streams Scalability

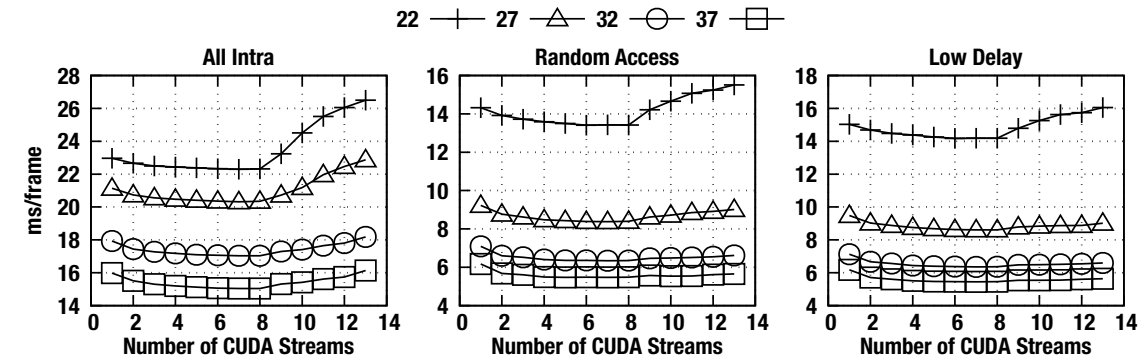
In order to evaluate the performance gains that were achieved by overlapping the GPU kernels and memory transfers, the input bitstreams were decoded by employing up to 13 CUDA Streams on the Titan GPU. The average processing time per frame (measured in ms) was obtained for each class and configuration (i.e., *All Intra*, *Random Access* and *Low Delay*). The obtained results are shown in Figure 5.2. Moreover, the achieved performance with each tested QP is presented as a set of points per each configuration.

As expected, for all considered classes and configurations, the resulting processing time per frame tends to decrease when the number of CUDA streams increases, until the minimum processing time per frame is reached. In particular, in the *All Intra* configuration, it is possible to observe that the achieved maximum performance corresponds to the use of 8 CUDA streams. Since the proposed GHEVC decoder bottleneck is the IP module (due to data dependencies between the blocks), the optimal number of CUDA streams corresponds to the minimum processing time imposed by the IP module. When more than eight CUDA streams are employed, the processing time increases mainly due to three factors:

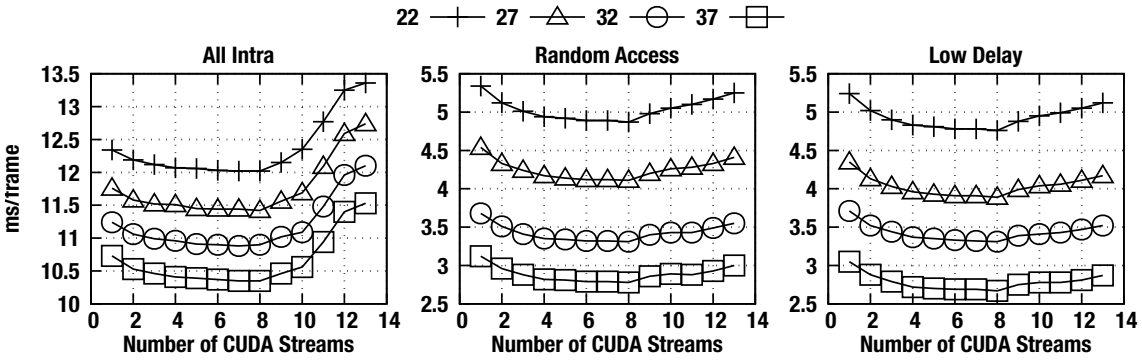
- **Used bandwidth**: when multiple streams are executed, the amount of data to be processed has to be divided accordingly, in order to support independent memory transfers and kernel executions in different streams. However, a larger number of smaller memory transfers may result in an inefficient use of the PCIe bandwidth.

- **Kernel overhead**: when launching a large number of kernels, the contribution of the time overhead associated with each kernel launch may decrease the overall performance.

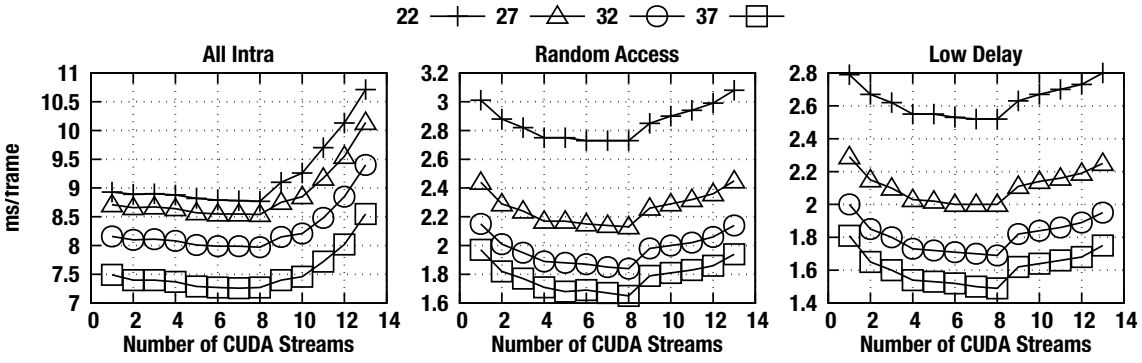
- **Occupancy**: whenever the kernels consume more resources than the GPU can provide, the amount of simultaneously running kernels is limited, resulting in a serialized execution of kernels.



(a) Class S – 3840×2160.



(b) Class A – 2560×1600.



(c) Class B – 1920×1080.

Figure 5.2: Evaluation of the performance scalability with the number of CUDA Streams for *All Intra*, *Random Access* and *Low Delay* configurations of a) Class S, b) Class A and c) Class B.

When the *Random Access* and *Low Delay* configurations are considered, the best performance is also achieved for 8 CUDA Streams. This is easily observed for *Class A* and *Class B* video sequences, depicted in Figure 5.2(b) and Figure 5.2(c), respectively. For *Class S* bitstreams, the IP module is not dominant for high QP values, as it can be observed in Figure 5.1(a). In this case, the minimum processing time is achieved for a number of streams higher than eight (for QP 32 and 37), since the MC module is the most time consuming in the proposed GHEVC decoder (see Figure 5.1(a) and Figure 5.2(a)). Hence, since the processing times per frame in those specific cases are very similar, 8 CUDA Streams will be considered for the subsequent experimental evaluation.

5.4 Comparison with Previous Intra GHEVC

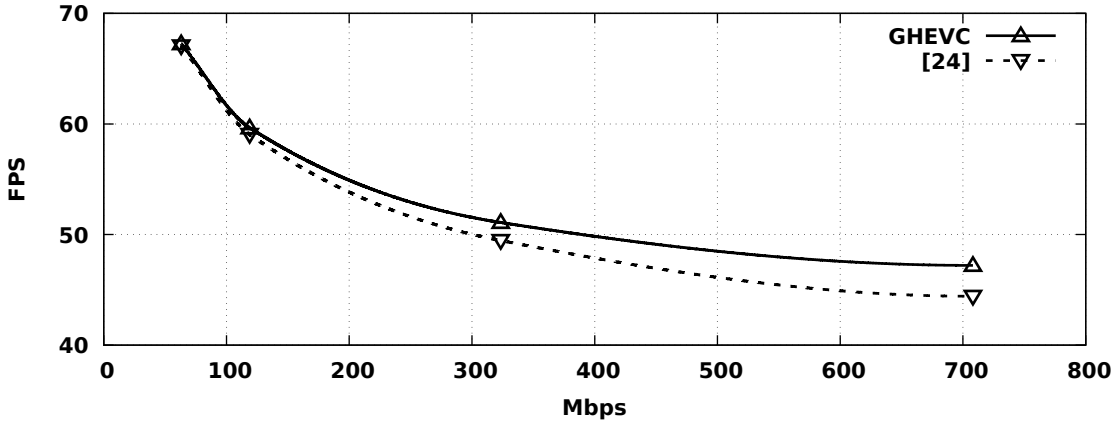
Figure 5.3 presents the performance improvement of the herein proposed GHEVC decoder over [24]. Since the implementation proposed in [24] refers to a HEVC intra decoder, only the *All Intra* configuration was considered in this evaluation. The experimental values were obtained with the NVIDIA Titan GPU by using 8 CUDA Streams, since this is the best setup for both HEVC decoders.

The presented frame rate values (frames per second (FPS)) were obtained by averaging the performance for all considered tested sequences for the different classes and QP configurations (from 22 to 37). Likewise, the input bitrate (corresponding to each QP) is also an average of the measured bitrate in megabit per second (Mbps) for the sequences within a class, which is obtained by multiplying the encoded bits/frame (for a specific QP and configuration) and the original frame rate (in FPS).

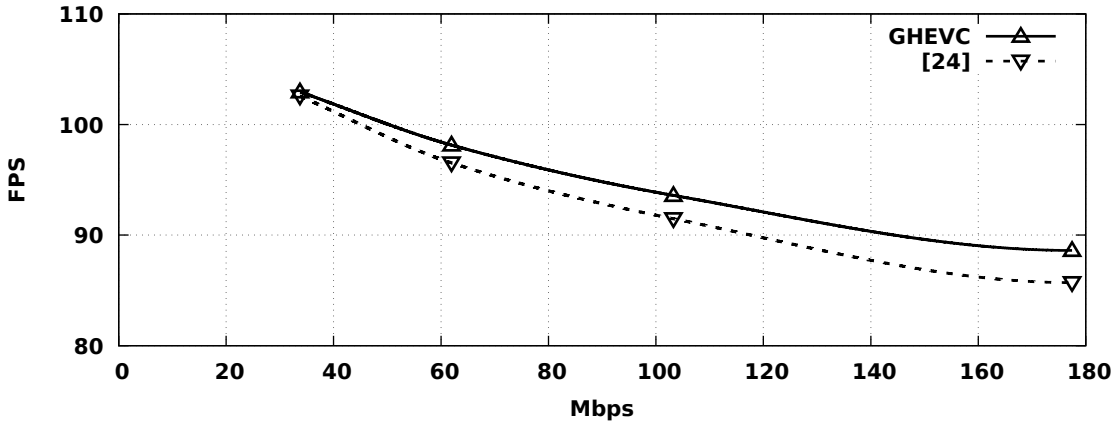
As it can be observed, the performance of the herein proposed GHEVC decoder is superior to the one that was obtained in [24] for all considered resolutions. When comparing the performance across different classes, the performance improvement of the proposed GHEVC decoder is higher for *Class B* (see Figure 5.3(c)). In this case, the obtained improvement is the result of a more efficient load balancing across the SMs, which becomes apparent due to the low parallelism level (i.e., wavefront size).

In [24], a ThB with eight warps is responsible for performing the intra prediction of a 64-pixel row of the frame, which, in a smaller wavefront size, implies that most of the SMs are idle during the IP kernel execution. In contrast, the newly proposed GHEVC decoder distributes a 64-pixel row of the frame across eight different ThBs. At the end, the execution time of the proposed GHEVC decoder is 6% faster than [24] in *Class B* video sequences.

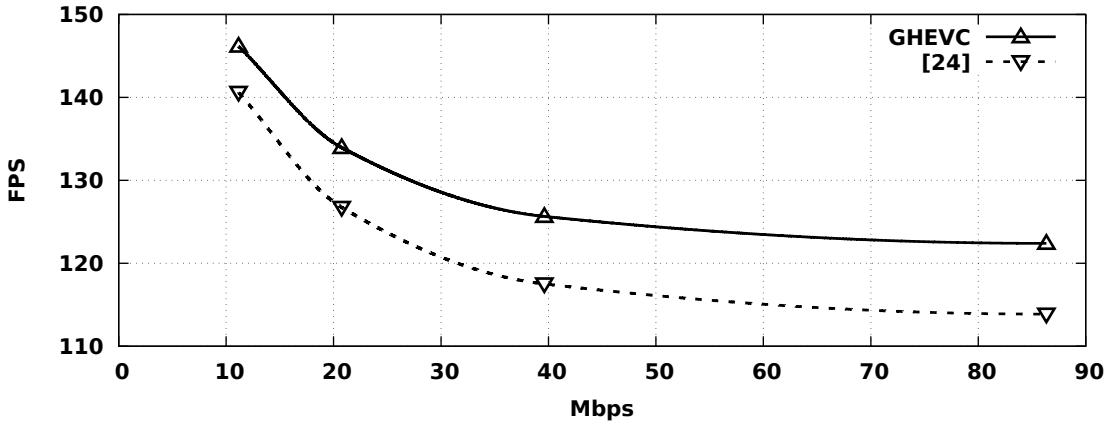
For *Class S* and *Class A*, a larger wavefront size, in comparison to the one in *Class B*, reduces the effect of the proposed load balancing in GHEVC, since there are less idle SMs in [24] during the IP kernel execution. Nevertheless, the proposed GHEVC decoder still provides slightly higher



(a) Class S – 3840×2160.



(b) Class A – 2560×1600.



(c) Class B – 1920×1080.

Figure 5.3: Overall performance of the herein proposed GHEVC and [24] on the Titan GPU for *All Intra* configuration in a) Class S, b) Class A and c) Class B.

performance than [24] (see Figure 5.3(a) and Figure 5.3(b)).

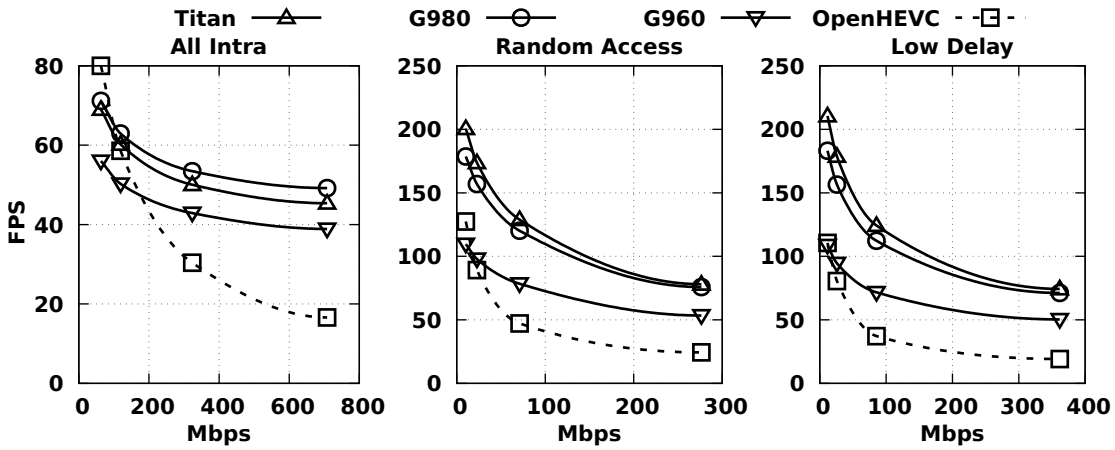
Within a single class, the performance improvement provided by the GHEVC decoder is greater for higher bitrates. This can be explained by the fact that, at lower bitrates, the input bitstream mostly includes larger prediction blocks (i.e., larger PU sizes), which implies less block dependencies to check and less idle SMs in both decoders.

5.5 GHEVC Decoding Performance

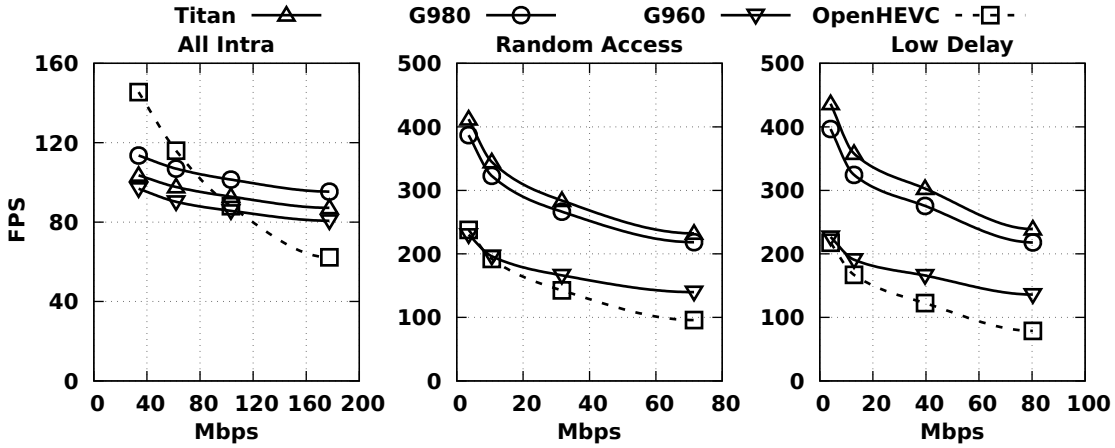
Figure 5.4 presents the conducted performance evaluation of the proposed GHEVC decoder, when compared with the OpenHEVC [112] CPU-based decoder executed with four threads on the Intel i7-6700K CPU @ 4.00GHz. The OpenHEVC decoder was chosen for the baseline comparison reference, although it is not a GPU-based HEVC decoder. Nevertheless, when considering real-time capability, it is the most commonly used open-source implementation in the literature. Moreover, as it was stated in Chapter 3, it was not possible to provide a fair and direct comparison with existing GPU-based HEVC decoders, since they are either exploiting a dedicated GPU decoding hardware or enough information is not provided since they are closed source. The presented performance in Figure 5.4 was obtained with three different GPUs from NVIDIA Maxwell architecture (Titan, G980 and G960) and corresponds to the resulting average frame rate (FPS) across all tested sequences, for each class and QP configuration (from 22 to 37). Furthermore, the presented bitrate (in Mbps) is an average bitrate for all video sequences within a class for each considered QP value.

As expected, both decoders (GHEVC and OpenHEVC) decrease the frame rate when the frame resolution is increased in all presented configurations, on account to the greater amount of data to be processed. Nevertheless, the proposed GHEVC decoder achieves higher frame rates: up to 69, 200 e 210 FPS of *Class S* in the Titan GPU for the *All Intra*, *Random Access* and *Low Delay* configurations, respectively (see Figure 5.4(a)). In fact, it can be observed that the proposed GHEVC decoder outperforms the OpenHEVC for the majority of the considered setups. The only exceptions are observed for the *All Intra* configuration for lower bitrates. In those cases, the strict data dependencies in the IP module do not allow fully exploiting the GPU capabilities.

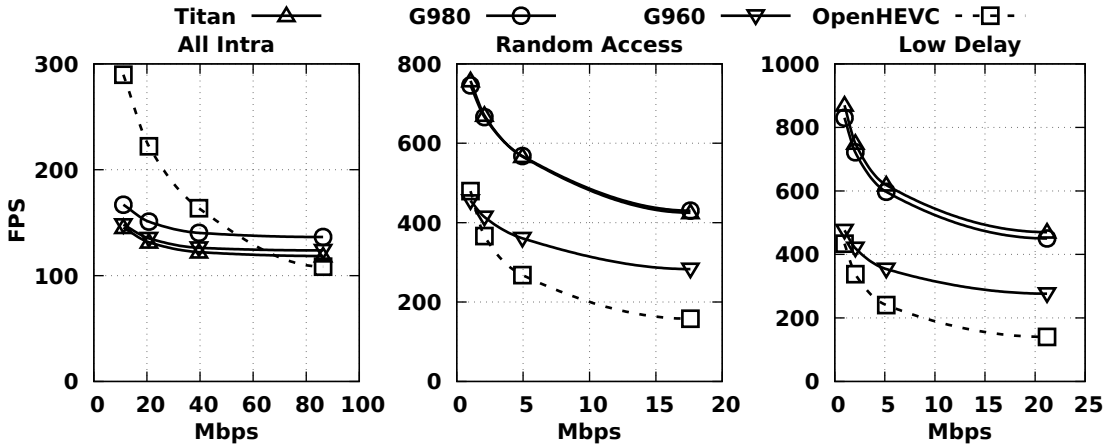
When looking at the GHEVC results, it can be observed that the G980 GPU performance is slightly higher than the Titan GPU performance in the *All Intra* configuration, although the latter one owns 50% more CUDA cores than the former. In fact, both GPU devices share the same architecture, with 128 CUDA cores per SM. However, while the Titan GPU has 24 SMs, the G980 GPU has only 16 SMs (see Table 5.2). On the other hand, the G980 GPU has a higher core clock frequency (1177 MHz) than the Titan GPU (1000 MHz). As a result, a greater number of SMs to execute kernels with a low degree of data parallelism, a higher amount of memory accesses and synchronization points (such as the IP kernel), may not necessarily provide performance



(a) Class S – 3840×2160.



(b) Class A – 2560×1600.



(c) Class B – 1920×1080.

Figure 5.4: Evaluation of the GHEVC decoder performance using NVIDIA Maxwell GPUs over the OpenHEVC decoder (running on the CPU).

benefits. First, the overall utilization of the cores and SMs is still limited by the amount of intrinsic IP parallelism provided in the wavefront (i.e., inherent data dependencies, as referred in Chapter 4). Second, increasing the amount of parallel memory requests in flight (from all SMs) may influence the amount of L2 cache evictions. Finally, by coupling these two effects with a lower operating frequency of the GPU cores (slower dispatch rate of instructions) and a slower operating speed of private/shared memory levels, a memory bound kernel does not necessarily benefits from an increased number of SMs.

In fact, this behavior can be observed for all the considered GPU devices from Maxwell architecture (G960, G980 and Titan) in *All Intra* configuration. In Figure 5.4, all three GPUs achieve a very similar performance level. However, as soon as the share of the IP in the total execution time decreases (i.e., when the share of data-parallel MC is increased in the *Random Access* and *Low Delay* configurations – see Figure 5.1), the benefits of increasing the number of SMs are more observable; the best performance is achieved with Titan, followed by G980 and G960. It is also worth noting that the achieved performance gain in these configurations does not directly correspond to the increase in the number of SMs across different GPU devices, since the intra prediction still has a significant share in the total execution time, thus diminishing the overall gain of other data-parallel kernels.

The average frame rate obtained with the proposed GHEVC decoder for all tested video sequences and for all considered GPU devices is presented in Table 5.4. In this evaluation, six NVIDIA GPU devices were used (i.e., Maxwell and Kepler), from high performance to low-end GPUs (see Table 5.2). Table 5.4 also presents the average power consumption (in Watts) for each GPU, class and configuration.

As expected, the attained frame rate in all GPU devices is higher for lower resolution video sequences (e.g., *Class B*), due to the small amount of data to be processed. Within a single class, the obtained frame rate for a GPU is different across the tested sequences, because of the characteristics of each sequence bitstream (i.e., the amount of intra blocks, the amount of smaller PU partitions, motion characteristics etc). For example, in the *All Intra* configuration, the *SteamLocomotive* and the *Kimono* bitstreams provide the highest performance for *Class A* and *Class B*, since they have the greatest amount of larger PU partitions among the remaining video bitstreams (of the same classes). Nevertheless, for *Class S*, the reduced amount of smaller PU partitions in all tested sequences leads to a more balanced performance among the sequences (e.g., in Titan GPU, the obtained frame rate is between 51 to 62 FPS).

When the *Random Access* and the *Low Delay* configurations are considered, it is observed that the amount of intra predicted blocks in inter type frames is the most limiting performance factor. For example, the video bitstreams of sequences *DucksTakeOff*, *PeopleOnStreet* and *BasketballDrive* are those with the higher amount of intra predicted blocks within their classes, which led to the lower performance in *Class S*, *Class A* and *Class B*, respectively, for any GPU.

Table 5.4: Average performance (FPS) obtained per tested sequence with the proposed GHEVC decoder.

	Sequence	All Intra						Random Access						Low Delay					
		Titan	G980	G960	G780	K40c	G680	Titan	G980	G960	G780	K40c	G680	Titan	G980	G960	G780	K40c	G680
Class S	CrowdRun	51.0	55.1	44.6	48.8	37.9	8.2	137.2	126.1	80.2	57.3	43.9	15.0	143.3	127.9	78.3	56.9	43.7	13.4
	DucksTakeOff	58.2	60.0	46.9	54.0	42.3	9.9	108.6	100.6	67.0	53.1	41.0	15.4	96.9	86.3	55.8	46.3	36.1	14.1
	InToTree	61.9	63.9	49.3	58.2	45.5	11.3	165.4	150.3	95.3	66.2	50.5	21.8	165.3	147.9	92.4	66.2	50.8	21.1
	OldTownCross	56.5	61.1	48.9	53.0	41.1	8.5	181.5	164.3	102.5	71.1	54.3	23.3	192.8	169.9	103.8	70.9	54.3	23.7
	ParkJoy	53.2	55.9	45.3	50.7	39.5	9.3	133.6	123.2	79.4	57.1	43.6	15.9	136.5	121.9	76.1	54.9	42.2	14.3
	Average Power (W)	90.8	68.5	38.5	–	83.4	–	91.0	68.6	38.6	–	83.5	–	90.9	68.6	38.6	–	83.4	–
Class A	Traffic	79.4	88.2	77.5	76.0	57.1	10.7	411.0	380.8	223.4	151.0	111.7	38.3	461.9	411.6	228.1	155.0	114.8	38.8
	PeopleOnStreet	81.1	90.8	79.9	77.8	58.4	10.0	261.7	247.2	155.7	114.0	85.2	20.5	284.3	259.2	157.6	112.0	84.4	18.7
	Nebuta	97.9	105.3	85.7	93.0	72.3	18.6	272.7	248.8	148.8	118.7	90.3	37.6	256.3	229.0	136.2	112.6	86.2	37.1
	SteamLocomotive	123.0	133.2	110.4	115.1	87.5	20.5	327.8	317.9	204.5	148.8	110.1	40.5	333.7	314.6	197.3	144.1	107.2	38.7
	Average Power (W)	90.5	68.2	38.4	–	82.8	–	90.5	68.3	38.4	–	82.8	–	90.5	68.4	38.2	–	82.7	–
Class B	Kimono	156.0	176.9	152.8	146.4	112.7	25.4	632.6	614.5	378.2	258.6	195.0	56.7	703.5	651.3	375.3	266.2	200.5	56.0
	ParkScene	121.0	139.2	125.7	113.7	87.2	15.0	601.1	605.5	379.7	247.7	185.5	47.9	737.4	704.7	402.2	263.6	196.9	47.5
	Cactus	120.0	138.5	126.2	113.9	86.8	16.0	592.2	611.1	393.4	261.0	192.1	49.2	674.8	671.4	398.2	268.1	200.2	48.8
	BQTerrace	124.9	145.4	132.8	117.2	89.3	14.3	735.6	698.5	414.5	266.6	200.7	56.8	792.8	731.9	414.5	271.4	204.0	56.4
	BasketballDrive	124.9	143.4	129.3	116.4	89.0	18.2	462.3	482.8	325.8	218.1	162.5	44.0	477.4	490.1	317.8	218.9	163.9	41.9
	Average Power (W)	90.3	67.8	37.9	–	82.5	–	89.8	66.6	36.8	–	82.2	–	90.2	68.1	38.1	–	82.4	–

In what concerns the power consumption, the obtained measures (using NVIDIA nvprof tool [137]) within a single class only slightly vary for the same GPU across different configurations (see Table 5.4). Nevertheless, within a single configuration, the power consumption increases with the frame resolution for all GPUs. Among the available GPUs, the G960 Maxwell GPU is the one with the lowest power consumption (around 38.2 W), which outperforms the K40c Kepler GPU not only in performance, but also in terms of energy efficiency.

In general, from the performance point of view, it is observed that the proposed GHEVC decoder on NVIDIA Maxwell GPUs outperforms the one on Kepler devices. Even the decoding procedure running on the low-end G960 Maxwell GPU is faster than the G780 and K40c high-performance Kepler GPUs in most of the cases. Nevertheless, an average frame rate above 30 FPS is obtained in almost all the cases, except for the oldest G680 GPU. In particular, for the high-performance Titan Maxwell GPU, average frame rates of 56, 145 and 147 FPS in the *All Intra*, *Random Access* and *Low Delay* configurations, respectively, are observed for *Class S*.

5.6 Summary

In this chapter, the proposed GHEVC decoder was experimentally evaluated. Six NVIDIA GPUs from two different architectures, i.e., *Maxwell* and *Kepler*, have been employed to assess the proposed GHEVC decoder. These six considered GPU devices are a rather representative range from low-end to high performance GPUs. The experimental evaluation was organized in five sections: *i*) the GHEVC ThB configuration, to investigate the best number of warps per ThB for all GHEVC kernels; *ii*) the GHEVC profiling, in order to identify the most time consuming modules and bottlenecks; *iii*) the GHEVC scalability, to verify how the performance of the proposed decoder is affected into a multiple CUDA streams scenario; *iv*) the comparison with previous work, to show how much improvement could be reached with the new IP kernel; *v*) the GHEVC decoding performance, where the overall decoding performance is compared with a state-of-the-art HEVC decoder, i.e., OpenHEVC, as well as with different NVIDIA architectures.

6

Conclusions and Future Work

Contents

An efficient GPU-based HEVC decoder (GHEVC), exploiting the massively parallel processing capabilities of current state-of-the-art GPU accelerators was proposed in this dissertation. To circumvent the added complexity of the decoding procedure defined by HEVC, it was proposed an efficient parallelization of the most important modules of an HEVC decoder targeting GPU accelerator architectures, including de-quantization, inverse transform, motion compensation, intra prediction, de-blocking filter and sample adaptive offset. To attain such objective, the presented investigation extensively exploits both fine and coarse-grained parallelization in an integrated perspective, by re-designing the execution pattern of the involved modules, while simultaneously coping with their inherent computational complexity and strict data dependencies.

Moreover, the proposed GHEVC decoder provides a complete GPU-based HEVC decompression solution, allowing not only the decoding of intra frames, but also of inter frames, including even the processing of intra blocks inside inter frames. As a result, the proposed GHEVC decoder nowadays represents the most comprehensive approach for HEVC compliant video decoding, including all the procedures specified by the HEVC standard.

To better highlight the presented novelties, the main contributions of this dissertation are summarized as follows:

- All HEVC decoder procedures have been comprehensively redesigned to fully exploit the parallelism, to optimize the memory access and to increase the instruction throughput. The provided design for each module allows a fully exploitation of the GPU devices without breaking the HEVC standard compliance.
- An unification of all the GHEVC modules, which reinforces data sharing among different HEVC procedures by taking advantage of the GPU's memory hierarchy. For example, in the proposed DBF the boundary strength is directly calculated in the GPU and no other data is needed to be retrieved from the CPU, since all required input data (from the other modules) is already available in the GPU's memory. Moreover, since the decoded frames are entirely generated in the GPU, they are kept in the GPU memory as long as they are needed as a reference frame for the MC module.
- A frame-level parallel processing, where different parts of the frame are processed in parallel in the GPU. CUDA Streams and explicit synchronization points ensure the HEVC standard compliance, while taking advantage of overlapping memory transactions and kernel executions.

This set of contributions also led to the following publications:

- D. F. de Souza, N. Roma, and L. Sousa. Cooperative CPU+GPU deblocking filter parallelization for high performance HEVC video codecs. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4993–4997, May 2014. doi:10.1109/ICASSP.2014.6854552.

- D. F. de Souza, N. Roma, and L. Sousa. OpenCL parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms. In *22nd European Signal Processing Conference (EUSIPCO)*, pages 755–759, Sept. 2014.

- D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. Towards GPU HEVC intra decoding: Seizing fine-grain parallelism. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, June 2015. doi:10.1109/ICME.2015.7177515.

- D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. HEVC in-loop filters GPU parallelization in embedded systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 123–130, July 2015. doi:10.1109/SAMOS.2015.7363667.

- D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU-assisted HEVC intra decoder. *Journal of Real-Time Image Processing*, 12(2):531–547, 2016. ISSN 1861-8219. doi:10.1007/s11554-015-0519-1.

- D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU acceleration of the HEVC decoder inter prediction module. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1245–1249, Dec. 2015. doi:10.1109/GlobalS IP.2015.7418397.

- B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. Efficient HEVC decoder for heterogeneous CPU with GPU systems. In *2016 IEEE 18th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6, Sept. 2016. doi:10.1109/MMSP.2016.7813353.

- B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU parallelization of HEVC in-loop filters. *International Journal of Parallel Programming*, pages 1–21, 2017. ISSN 1573-7640. doi:10.1007/s10766-017-0488-z.

- D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GHEVC: An efficient HEVC decoder for graphics processing units. *IEEE Transactions on Multimedia*, 19(3):459–474, Mar. 2017. ISSN 1520-9210. doi:10.1109/TMM.2016.2625261.

- B. Wang, D. F. de Souza, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, A. Ilic, N. Roma, and L. Sousa. Highly parallel HEVC decoding for heterogeneous systems with CPU and GPU. *Signal Processing: Image Communication*, 2018.

The presented GHEVC decoder executes the whole decoding pipeline at the GPU, except for the entropy decoder module, which is kept on the CPU side due to its highly irregular execution pattern. With the considered decoding structure, the frames are completely decompressed in the GPU device and kept in the GPU memory for the subsequent inter frame predictions. All the

required data was carefully packed and managed in order to avoid stride GPU global memory accesses and minimize the memory transactions. Moreover, all the deblocking filter decisions are entirely performed on the GPU side, by manipulating the already existing data. Furthermore, to take the maximum advantage of CUDA Streams, each frame is divided into a set of regions that are processed by several streams cautiously updated in order to ensure the compliance with the HEVC standard.

A comprehensive profiling of all the GHEVC decoder modules identified the current design bottleneck, which is, as expected, the most data-dependent module, *Intra Prediction*. Moreover, an evaluation of the overlap between GPU computations and memory transfers provided an insightful knowledge on how the proposed decoder performs with CUDA streams. In the GHEVC decoder, eight CUDA Streams provided the best overall performance in all tested configurations.

When compared with the open-source OpenHEVC decoder (with four CPU threads), the proposed GHEVC decoder presents significant improvements in most application scenarios, by providing average frame rates of 145, 318 and 605 frames per second for *Ultra HD 4K*, *WQXGA* and *Full HD*, respectively, in the *Random Access* configuration. Finally, an evaluation of the GHEVC decoder with several GPU devices, from low-end to high performance from *Maxwell* and *Kepler* architectures, has also been performed. Such experimental results showed that real-time processing is achieved for most common GPU architectures and devices.

6.1 Future Work

Within the scope of this dissertation, the following topics can be considered for future work:

- Development of the GHEVC support on heterogeneous systems with multiple GPUs. Since there are no reference frames in the *All Intra* configuration, all frames of the sequence could be decompressed in parallel. In this case, each frame could be decompressed in a different GPU device. When considering configurations with inter type frames, different frames can also be processed in parallel according to their reference frames. Moreover, even if there are inter frame dependencies, parts of the frames can also be decompressed in parallel. For example, if half of the reference frame has been already decompressed, the first CTU rows can be processed by another GPU.

- Development of an OpenCL GHEVC version. Although the CUDA version of the GHEVC allows to deeply take advantage of NVIDIA GPUs, an OpenCL version would allow to explore other GPU architectures and devices. Another option is to port the code to the recently launched Heterogeneous-compute Interface for Portability (HIP) tool from AMD.

- Development of the GHEVC decoding support for the HEVC Range Extensions. Different chroma sampling formats, such as monochrome, 4:2:2 and 4:4:4, as well as increased

sample bit depths beyond 8 bits have not been considered in the proposed version of the GHEVC.

- Development of the GHEVC entropy decoder to support multithreading. Since the entropy decoder is the most sequential HEVC procedure, which could not be ported to the GPU, a multithread solution could speed up the overall process. Furthermore, high-level parallelization techniques such as Tiles and WPP could also be supported.

References

[1] Cisco. *Cisco Visual Networking Index: Forecast and Methodology, 2015–2020*. White Paper, June 2016.

[2] ITU-T VCEG and ISO/IEC MPEG. *Terms of Reference of the Joint Collaborative Team on Video Coding Standard Development*. Doc. VCEG-AM90, Jan. 2010.

[3] ITU-T VCEG and ISO/IEC MPEG. *Joint Call for Proposals on Video Compression Technology*. Doc. VCEG-AM91, Jan. 2010.

[4] JCT-VC. *High Efficient Video Coding (HEVC)*. ITU-T Recommendation H.265 and ISO/IEC 23008-2, ITU-T and ISO/IEC JTC 1, Apr. 2013.

[5] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221191.

[6] J.-R. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand. Comparison of the coding efficiency of video coding standards – including high efficiency video coding (HEVC). *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1669–1684, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221192.

[7] T. K. Tan, M. Mrak, V. Baroncini, and N. Ramzan. *HEVC verification test results*. Doc. JCTVC-Q0204, Mar. 2014.

[8] JVT. *Advanced Video Coding for Generic Audio-Visual Services*. ITU-T Recommendation H.264 and ISO/IEC 14496-10 Advanced Video Coding, May 2003 (and subsequent versions).

[9] G. J. Sullivan, P. N. Topiwala, and A. Luthra. The H.264/AVC advanced video coding standard: overview and introduction to the fidelity range extensions. In *Proc. SPIE*, volume 5558, pages 454–474, 2004. doi:10.1117/12.564457.

[10] F. Bossen, B. Bross, K. Suhring, and D. Flynn. HEVC complexity and implementation analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1685–1696, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221255.

[11] J. Vanne, M. Viitanen, T. D. Hamalainen, and A. Hallapuro. Comparative rate-distortion-complexity analysis of HEVC and AVC video codecs. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1885–1898, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2223013.

[12] C. C. Chi, M. Alvarez-Mesa, J. Lucas, B. Juurlink, and T. Schierl. Parallel HEVC decoding on multi- and many-core architectures. *Journal of Signal Processing Systems*, 71(3):247–260, June 2013. ISSN 1939-8018. doi:10.1007/s11265-012-0714-2.

[13] M. Wien. *Design and Specification*, pages 73–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_3.

[14] H. Schwarz, T. Schierl, and D. Marpe. *Block Structures and Parallelism Features in HEVC*, pages 49–90. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_3.

[15] M. Tikekar, C.-T. Huang, C. Juvekar, V. Sze, and A. Chandrakasan. *Decoder Hardware Architecture for HEVC*, pages 303–341. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_10.

[16] C. C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink, and T. Schierl. SIMD acceleration for HEVC decoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(5): 841–855, May 2015. ISSN 1051-8215. doi:10.1109/TCSVT.2014.2364413.

[17] S.-F. Tsai, C.-H. Tsai, and L.-G. Chen. *Encoder Hardware Architecture for HEVC*, pages 343–375. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_11.

[18] M. Wien. *Profiles, Tiers, and Levels*, pages 283–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_11.

[19] Y. Duan, J. Sun, L. Yan, K. Chen, and Z. Guo. Novel efficient HEVC decoding solution on general-purpose processors. *IEEE Transactions on Multimedia*, 16(7):1915–1928, Nov. 2014. ISSN 1520-9210. doi:10.1109/TMM.2014.2337834.

[20] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008. ISSN 0272-1732. doi:10.1109/MM.2008.31.

[21] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., Waltham, MA, USA, 2nd edition, 2013. ISBN 0124159923, 9780124159921.

[22] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GHEVC: An efficient HEVC decoder for graphics processing units. *IEEE Transactions on Multimedia*, 19(3):459–474, Mar. 2017. ISSN 1520-9210. doi:10.1109/TMM.2016.2625261.

[23] B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU parallelization of HEVC in-loop filters. *International Journal of Parallel Programming*, pages 1–21, 2017. ISSN 1573-7640. doi:10.1007/s10766-017-0488-z.

[24] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU-assisted HEVC intra decoder. *Journal of Real-Time Image Processing*, 12(2):531–547, 2016. ISSN 1861-8219. doi:10.1007/s11554-015-0519-1.

[25] B. Wang, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. Efficient HEVC decoder for heterogeneous CPU with GPU systems. In *2016 IEEE 18th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6, Sept. 2016. doi:10.1109/MMSP.2016.7813353.

[26] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. GPU acceleration of the HEVC decoder inter prediction module. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 1245–1249, Dec. 2015. doi:10.1109/GlobalS IP.2015.7418397.

[27] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. HEVC in-loop filters GPU parallelization in embedded systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*, pages 123–130, July 2015. doi:10.1109/SAMOS.2015.7363667.

[28] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa. Towards GPU HEVC intra decoding: Seizing fine-grain parallelism. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, June 2015. doi:10.1109/ICME.2015.7177515.

[29] D. F. de Souza, N. Roma, and L. Sousa. OpenCL parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms. In *22nd European Signal Processing Conference (EUSIPCO)*, pages 755–759, Sept. 2014.

[30] D. F. de Souza, N. Roma, and L. Sousa. Cooperative CPU+GPU deblocking filter parallelization for high performance HEVC video codecs. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4993–4997, May 2014. doi:10.1109/ICASSP.2014.6854552.

[31] B. Wang, D. F. de Souza, M. Alvarez-Mesa, C. C. Chi, B. Juurlink, A. Ilic, N. Roma, and L. Sousa. Highly parallel HEVC decoding for heterogeneous systems with CPU and GPU. *Signal Processing: Image Communication*, 2018.

[32] S. Mittal and J. S. Vetter. A survey of CPU-GPU heterogeneous computing techniques. *ACM Computing Surveys*, 47(4):69:1–69:35, July 2015. ISSN 0360-0300. doi:10.1145/2788396.

[33] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30(2):56–69, Mar. 2010. ISSN 0272-1732. doi:10.1109/MM.2010.41.

[34] 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*, 15(5): 685–693, May 2005. ISSN 1051-8215. doi:10.1109/TCSVT.2005.846440.

[35] S. Yamagiwa and L. Sousa. Caravela: A novel stream-based distributed computing environment. *Computer*, 40(5):70–77, May 2007. ISSN 0018-9162. doi:10.1109/MC.2007.161.

[36] NVIDIA. *NVIDIA® CUDA™ Compute Unified Device Architecture Programming Guide*, version 1.0: June 2007 (and subsequent editions).

[37] A. Munshi. The OpenCL specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314, Aug. 2009. doi:10.1109/HOTCHIPS.2009.7478342.

[38] NVIDIA. *OpenCL Programming Guide for the CUDA Architecture*, version 2.3: August 2009.

[39] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. ISSN 0018-9219. doi:10.1109/JPROC.2008.917757.

[40] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov. Parallel computing experiences with CUDA. *IEEE Micro*, 28(4): 13–27, July 2008. ISSN 0272-1732. doi:10.1109/MM.2008.57.

[41] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008. ISSN 1542-7730. doi:10.1145/1365490.1365500.

[42] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008. ISBN 0123744938, 9780123744937.

[43] Advanced Micro Devices Inc. *AMD Accelerated Parallel Processing OpenCL™ Programming Guide*, version 2.7: Nov. 2013.

[44] NVIDIA. *NVIDIA® CUDA™ Compute Unified Device Architecture C Best Practices Guide*, version 7.5: Sept. 2015.

[45] I.-K. Kim, J. Min, T. Lee, W.-J. Han, and J. Park. Block partitioning structure in the HEVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1697–1706, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2223011.

[46] M. Budagavi, A. Fuldseth, G. Bjøntegaard, V. Sze, and M. Sadafale. Core transform design in the high efficiency video coding (HEVC) standard. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):1029–1041, Dec. 2013. ISSN 1932-4553. doi:10.1109/JSTSP.2013.2270429.

[47] K. Ugur, A. Alshin, E. Alshina, F. Bossen, W.-J. Han, J.-H. Park, and J. Lainema. Motion compensated prediction and interpolation filter design in H.265/HEVC. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):946–956, Dec. 2013. ISSN 1932-4553. doi:10.1109/JSTSP.2013.2272771.

[48] J. Lainema, F. Bossen, W. J. Han, J. Min, and K. Ugur. Intra coding of the HEVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1792–1801, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221525.

[49] J. Ohm. *Multimedia Communication Technology*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-01249-8.

[50] A. Norkin, G. Bjøntegaard, A. Fuldseth, M. Narroschke, M. Ikeda, K. Andersson, M. Zhou, and G. Van der Auwera. HEVC deblocking filter. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1746–1754, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2223053.

[51] C.-M. Fu, E. Alshina, A. Alshin, Y.-W. Huang, C.-Y. Chen, C.-Y. Tsai, C.-W. Hsu, S.-M. Lei, J.-H. Park, and W.-J. Han. Sample adaptive offset in the HEVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1755–1764, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221529.

[52] V. Sze and D. Marpe. *Entropy Coding in HEVC*, pages 209–274. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_8.

[53] M. Wien. *Entropy Coding*, pages 251–282. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_10.

[54] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, July 2003. ISSN 1051-8215. doi:10.1109/TCSVT.2003.815173.

[55] V. Sze and M. Budagavi. High throughput CABAC entropy coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1778–1791, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221526.

[56] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl. Parallel scalability and efficiency of HEVC parallelization approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1827–1838, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2223056.

[57] K. Misra, A. Segall, M. Horowitz, S. Xu, A. Fuldseth, and M. Zhou. An overview of tiles in HEVC. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):969–977, Dec. 2013. ISSN 1932-4553. doi:10.1109/JSTSP.2013.2271451.

[58] T. Nguyen, P. Helle, M. Winken, B. Bross, D. Marpe, H. Schwarz, and T. Wiegand. Transform coding techniques in HEVC. *IEEE Journal of Selected Topics in Signal Processing*, 7(6): 978–989, Dec. 2013. ISSN 1932-4553. doi:10.1109/JSTSP.2013.2278071.

[59] J. Sole, R. Joshi, N. Nguyen, T. Ji, M. Karczewicz, G. Clare, F. Henry, and A. Duenas. Transform coefficient coding in HEVC. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1765–1777, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2223055.

[60] M. Budagavi, A. Fuldseth, and G. Bjøntegaard. *HEVC Transform and Quantization*, pages 141–169. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_6.

[61] M. Zhou, W. Gao, M. Jiang, and H. Yu. HEVC lossless coding and improvements. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1839–1843, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2221524.

[62] M. Wien. *Residual Coding*, pages 205–227. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_8.

[63] C. G. Kim and Y. S. Choi. A high performance parallel DCT with OpenCL on heterogeneous computing environment. *Multimedia Tools and Applications*, 64(2):475–489, 2013. ISSN 1380-7501. doi:10.1007/s11042-012-1028-x.

[64] L.-P. He and S. Goto. A high parallel way for processing IQ/IT part of HEVC decoder based on GPU. In *Intelligent Signal Processing and Communication Systems (ISPACS), 2014 International Symposium on*, pages 211–215, Dec. 2014. doi:10.1109/ISPACS.2014.7024454.

[65] K. Panusopone, L. Wang, and X. Fang. Implicit transform unit partitioning in HEVC. In *Picture Coding Symposium (PCS), 2013*, pages 213–216, Dec. 2013. doi:10.1109/PCS.2013.6737721.

[66] H. Fan, R. Wang, L. Ding, X. Xie, H. Jia, and W. Gao. Hybrid zero block detection for high efficiency video coding. *IEEE Transactions on Multimedia*, 18(3):537–543, Mar. 2016. ISSN 1520-9210. doi:10.1109/TMM.2016.2515365.

[67] K. Lee, H.-J. Lee, J. Kim, and Y. Choi. A novel algorithm for zero block detection in high efficiency video coding. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):1124–1134, Dec. 2013. ISSN 1932-4553. doi:10.1109/JSTSP.2013.2272772.

[68] L. A. Sousa. General method for eliminating redundant computations in video coding. *Electronics Letters*, 36(4):306–307, Feb. 2000. ISSN 0013-5194. doi:10.1049/el:20000272.

[69] H. Yong, R. Wang, W. Wang, Z. Wang, S. Dong, B. Han, and W. Gao. Acceleration of HEVC transform and inverse transform on ARM NEON platform. In *Intelligent Signal Processing and Communications Systems (ISPACS), 2013 International Symposium on*, pages 169–173, Nov. 2013. doi:10.1109/ISPACS.2013.6704541.

[70] M. Budagavi and V. Sze. Unified forward+inverse transform architecture for HEVC. In *2012 19th IEEE International Conference on Image Processing*, pages 209–212, Sept. 2012. doi:10.1109/ICIP.2012.6466832.

[71] L. Hong, W. He, H. Zhu, and Z. Mao. A cost effective 2-D adaptive block size IDCT architecture for HEVC standard. In *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 1290–1293, Aug. 2013. doi:10.1109/MWSCAS.2013.6674891.

[72] M. Tikekar, C. T. Huang, V. Sze, and A. Chandrakasan. Energy and area-efficient hardware implementation of HEVC inverse transform and dequantization. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 2100–2104, Oct. 2014. doi:10.1109/ICIP.2014.7025421.

[73] T. Dias, N. Roma, and L. Sousa. High performance IP core for HEVC quantization. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2828–2831, May 2015. doi:10.1109/ISCAS.2015.7169275.

[74] J. Goebel, G. Paim, L. Agostini, B. Zatt, and M. Porto. An HEVC multi-size DCT hardware with constant throughput and supporting heterogeneous CUs. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2202–2205, May 2016. doi:10.1109/ISCAS.2016.7539019.

[75] M. Wien. *Inter Prediction*, pages 179–203. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_7.

[76] B. Bross, P. Helle, H. Lakshman, and K. Ugur. *Inter-Picture Prediction in HEVC*, pages 113–140. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_5.

[77] Y. Yuan, I. K. Kim, X. Zheng, L. Liu, X. Cao, S. Lee, M. S. Cheon, T. Lee, Y. He, and J. H. Park. Quadtree based nonsquare block structure for inter frame coding in high efficiency

video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12): 1707–1719, Dec. 2012. ISSN 1051-8215. doi:10.1109/TCSVT.2012.2223037.

[78] G. Cebrián-Márquez, J. L. Hernández-Losada, J. L. Martínez, P. Cuenca, M. Tang, and J. Wen. Accelerating HEVC using heterogeneous platforms. *The Journal of Supercomputing*, 71(2):613–628, 2015. ISSN 1573-0484. doi:10.1007/s11227-014-1313-8.

[79] S. Radicke, J.-U. Hahn, Q. Wang, and C. Grecos. Bi-predictive motion estimation for HEVC on a graphics processing unit (GPU). *IEEE Transactions on Consumer Electronics*, 60(4): 728–736, Nov. 2014. ISSN 0098-3063. doi:10.1109/TCE.2014.7027349.

[80] A. Ilic, S. Momcilovic, N. Roma, and L. Sousa. Adaptive scheduling framework for real-time video encoding on heterogeneous systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(3):597–611, Mar. 2016. ISSN 1051-8215. doi:10.1109/TCSVT.2015.2402893.

[81] B. Wang, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink. Parallel H.264/AVC motion compensation for GPUs using OpenCL. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(3):525–531, Mar. 2015. ISSN 1051-8215. doi:10.1109/TCSVT.2014.2344512.

[82] Z. Guo, D. Zhou, and S. Goto. An optimized MC interpolation architecture for HEVC. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1117–1120, Mar. 2012. doi:10.1109/ICASSP.2012.6288083.

[83] C. M. Diniz, M. Shafique, S. Bampi, and J. Henkel. High-throughput interpolation hardware architecture with coarse-grained reconfigurable datapaths for HEVC. In *2013 IEEE International Conference on Image Processing*, pages 2091–2095, Sept. 2013. doi:10.1109/ICIP.2013.6738431.

[84] S. Wang, D. Zhou, J. Zhou, T. Yoshimura, and S. Goto. VLSI implementation of HEVC motion compensation with distance biased direct cache mapping for 8K UHDTV applications. *IEEE Transactions on Circuits and Systems for Video Technology*, PP(99):1–1, 2015. ISSN 1051-8215. doi:10.1109/TCSVT.2015.2511858.

[85] W. Penny, G. Paim, M. Porto, L. Agostini, and B. Zatt. Real-time architecture for HEVC motion compensation sample interpolator for UHD videos. In *2015 28th Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–6, Aug. 2015.

[86] M. Wien. *Intra Prediction*, pages 161–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_6.

[87] J. Lainema and W.-J. Han. *Intra-Picture Prediction in HEVC*, pages 91–112. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_4.

[88] E. Kalali, Y. Adibelli, and I. Hamzaoglu. A high performance and low energy intra prediction hardware for HEVC video decoding. In *Design and Architectures for Signal and Image Processing (DASIP), 2012 Conference on*, pages 1–8, Oct. 2012.

[89] C.-T. Huang, M. Tikekar, and A. P. Chandrakasan. Memory-hierarchical and mode-adaptive HEVC intra prediction architecture for Quad Full HD video decoding. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(7):1515–1525, July 2014. ISSN 1063-8210. doi:10.1109/TVLSI.2013.2275571.

[90] J. Zhou, D. Zhou, H. Sun, and S. Goto. VLSI architecture of HEVC intra prediction for 8K UHDTV applications. In *2014 IEEE International Conference on Image Processing (ICIP)*, pages 1273–1277, Oct. 2014. doi:10.1109/ICIP.2014.7025254.

[91] M. Wien. *In-Loop Filtering*, pages 229–250. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-44276-0. doi:10.1007/978-3-662-44276-0_9.

[92] A. Norkin, K. Andersson, A. Fuldseth, and G. Bjøntegaard. HEVC deblocking filtering and decisions. In *Proc. SPIE*, volume 8499, pages 849912–849912–8, 2012. doi:10.1117/12.970326.

[93] A. Norkin, C.-M. Fu, Y.-W. Huang, and S. Lei. *In-Loop Filters in HEVC*, pages 171–208. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06895-4. doi:10.1007/978-3-319-06895-4_7.

[94] A. M. Kotra, M. Raulet, and O. Deforges. Comparison of different parallel implementations for deblocking filter of HEVC. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2721–2725, 2013. doi:10.1109/ICASSP.2013.6638151.

[95] H. Jo, S. Park, and D. Sim. Parallelized deblocking filtering of HEVC decoders based on complexity estimation. *Journal of Real-Time Image Processing*, 12(2):369–382, 2016. ISSN 1861-8219. doi:10.1007/s11554-015-0556-9.

[96] A. F. Eldeken, R. M. Dansereau, M. M. Fouad, and G. I. Salama. High throughput parallel scheme for HEVC deblocking filter. In *Image Processing (ICIP), 2015 IEEE International Conference on*, pages 1538–1542, Sept. 2015. doi:10.1109/ICIP.2015.7351058.

[97] W. Jiang, H. Mei, F. Lu, H. Jin, L. T. Yang, B. Luo, and Y. Chi. A novel parallel deblocking filtering strategy for HEVC/H.265 based on GPU. *Concurrency and Computation: Practice and Experience*, 2016. ISSN 1532-0634. doi:10.1002/cpe.3751. CPE-15-0134.R1.

[98] E. Ozcan, Y. Adibelli, and I. Hamzaoglu. A high performance deblocking filter hardware for high efficiency video coding. *IEEE Transactions on Consumer Electronics*, 59(3):714–720, Aug. 2013. ISSN 0098-3063. doi:10.1109/TCE.2013.6626260.

[99] W. Shen, Q. Shang, S. Shen, Y. Fan, and X. Zeng. A high-throughput VLSI architecture for deblocking filter in HEVC. In *2013 IEEE International Symposium on Circuits and Systems (ISCAS2013)*, pages 673–676, May 2013. doi:10.1109/ISCAS.2013.6571936.

[100] C. M. Diniz, M. Shafique, F. V. Dalcin, S. Bampi, and J. Henkel. A deblocking filter hardware architecture for the high efficiency video coding standard. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1509–1514, Mar. 2015. doi:10.7873/DATE.2015.0856.

[101] W. Zhou, J. Zhang, X. Zhou, Z. Liu, and X. Liu. A high-throughput and multi-parallel VLSI architecture for HEVC deblocking filter. *IEEE Transactions on Multimedia*, 18(6):1034–1047, June 2016. ISSN 1520-9210. doi:10.1109/TMM.2016.2537217.

[102] Y. Wang, X. Guo, Y. Lu, X. Fan, and D. Zhao. GPU-based optimization for sample adaptive offset in HEVC. In *2016 IEEE International Conference on Image Processing (ICIP)*, pages 829–833, Sept. 2016. doi:10.1109/ICIP.2016.7532473.

[103] W. Zhang and C. Guo. Design and implementation of parallel algorithms for sample adaptive offset in HEVC based on GPU. In *2016 Sixth International Conference on Information Science and Technology (ICIST)*, pages 181–187, May 2016. doi:10.1109/ICIST.2016.7483407.

[104] F. Luo, S. Wang, N. Zhang, S. Ma, and W. Gao. GPU based sample adaptive offset parameter decision and perceptual optimization for HEVC. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2687–2690, May 2016. doi:10.1109/ISCAS.2016.7539147.

[105] S. Park and K. Ryoo. The hardware design of effective SAO for HEVC decoder. In *2013 IEEE 2nd Global Conference on Consumer Electronics (GCCE)*, pages 303–304, Oct. 2013. doi:10.1109/GCCE.2013.6664837.

[106] I. Hautala, J. Boutellier, J. Hannuksela, and O. Silvén. Programmable low-power multicore coprocessor architecture for HEVC/H.265 in-loop filtering. *IEEE Transactions on Circuits and Systems for Video Technology*, 25(7):1217–1230, July 2015. ISSN 1051-8215. doi:10.1109/TCSVT.2014.2369744.

[107] I. Hautala, J. Boutellier, and O. Siiven. Programmable 28nm coprocessor for HEVC/H.265 in-loop filters. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1570–1573, May 2016. doi:10.1109/ISCAS.2016.7538863.

[108] M. Mody, N. Nandan, and T. Hideo. High throughput VLSI architecture supporting HEVC loop filter for ultra HDTV. In *2013 IEEE Third International Conference on Consumer Electronics – Berlin (ICCE-Berlin)*, pages 54–57, Sept. 2013. doi:10.1109/ICCE-Berlin.2013.6698026.

[109] J. Zhu, D. Zhou, G. He, and S. Goto. A combined SAO and de-blocking filter architecture for HEVC video decoder. In *2013 IEEE International Conference on Image Processing*, pages 1967–1971, Sept. 2013. doi:10.1109/ICIP.2013.6738405.

[110] D. Flynn, D. Marpe, M. Naccari, T. Nguyen, C. Rosewarne, K. Sharman, J. Sole, and J. Xu. Overview of the range extensions for the HEVC standard: Tools, profiles, and performance. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(1):4–19, Jan. 2016. ISSN 1051-8215. doi:10.1109/TCSVT.2015.2478707.

[111] Joint Collaborative Team on Video Coding (JCT-VC) of ISO/IEC MPEG and ITU-T VCEG. Subversion repository for the HEVC Test Model (HM), accessed: 2016, Oct. 01. URL `https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/`.

[112] OpenHEVC. Open source HEVC decoder (OpenHEVC), accessed: 2016, Oct. 01. URL `https://github.com/OpenHEVC/openHEVC`.

[113] L. Yan, Y. Duan, J. Sun, and Z. Guo. Implementation of HEVC decoder on x86 processors with SIMD optimization. In *Visual Communications and Image Processing (VCIP), 2012 IEEE*, pages 1–6, Nov. 2012. doi:10.1109/VCIP.2012.6410845.

[114] L. Yan, Y. Duan, J. Sun, and Z. Guo. An optimized real-time multi-thread HEVC decoder. In *Visual Communications and Image Processing (VCIP), 2012 IEEE*, pages 1–1, Nov. 2012. doi:10.1109/VCIP.2012.6410857.

[115] H. Jo, D. Sim, and B. Jeon. Hybrid parallelization for HEVC decoder. In *Image and Signal Processing (CISP), 2013 6th International Congress on*, volume 01, pages 170–175, Dec. 2013. doi:10.1109/CISP.2013.6743980.

[116] J. Jeong, J. Choi, and S. Ha. Parallelization and performance prediction for HEVC UHD real-time software decoding. In *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pages 40–49, Oct. 2014. doi:10.1109/ESTIMedia.2014.6962344.

[117] B. Han, R. Wang, Z. Wang, S. Dong, W. Wang, and W. Gao. HEVC decoder acceleration on multi-core X86 platform. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 7353–7357, May 2014. doi:10.1109/ICASSP.2014.6855028.

[118] M. Bariani, P. Lambruschini, M. Raggio, and L. Pezzoni. An optimized software implementation of the hevc/h.265 video decoder. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pages 77–82, Jan. 2014. doi:10.1109/CCNC.2014.7056307.

[119] C. C. Chi, M. Alvarez-Mesa, and B. Juurlink. Low-power high-efficiency video decoding using general-purpose processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):56:1–56:25, Jan. 2015. ISSN 1544-3566. doi:10.1145/2685551.

[120] E. Nogues, J. Heulot, G. Herrou, L. Robin, M. Pelcat, D. Menard, E. Raffin, and W. Hamidouche. Efficient DVFS for low power HEVC software decoder. *Journal of Real-Time Image Processing*, pages 1–16, 2016. ISSN 1861-8219. doi:10.1007/s11554-016-0624-9.

[121] R. Rodríguez-Sánchez and E. S. Quintana-Ortí. Architecture-aware optimization of an HEVC decoder on asymmetric multicore processors. *Journal of Real-Time Image Processing*, pages 1–14, 2016. ISSN 1861-8219. doi:10.1007/s11554-016-0606-y.

[122] R. Arzumanyan, A. Fartukov, and Jeik Kim. Study of task scheduling for HEVC video decoder in heterogeneous computational environment. In *2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE)*, pages 577–579, Oct. 2014. doi:10.1109/GCCE.2014.7031103.

[123] M. Chavarrias, F. Pescador, M. J. Garrido, E. Juárez, and M. Raulet. A DSP-based HEVC decoder implementation using an actor language dataflow model. *IEEE Transactions on Consumer Electronics*, 59(4):839–847, Nov. 2013. ISSN 0098-3063. doi:10.1109/TCE.2013.6689697.

[124] M. Chavarrías, F. Pescador, M. J. Garrido, E. Juárez, and C. Sanz. A multicore DSP HEVC decoder using an actorbased dataflow model and OpenMP. *IEEE Transactions on Consumer Electronics*, 61(2):236–244, May 2015. ISSN 0098-3063. doi:10.1109/TCE.2015.7150599.

[125] D. Engelhardt, J. Möller, J. Hahlbeck, and B. Stabernack. FPGA implementation of a Full HD real-time HEVC main profile decoder. *IEEE Transactions on Consumer Electronics*, 60(3): 476–484, Aug. 2014. ISSN 0098-3063. doi:10.1109/TCE.2014.6937333.

[126] C. T. Huang, M. Tikekar, C. Juvekar, V. Sze, and A. Chandrakasan. A 249Mpixel/s HEVC video-decoder chip for Quad Full HD applications. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 162–163, Feb. 2013. doi:10.1109/ISSCC.2013.6487682.

[127] C. C. Ju, T. M. Liu, Y. C. Chang, C. M. Wang, H. M. Lin, C. Y. Cheng, C. C. Chen, M. H. Chiu, S. J. Wang, P. Chao, M. J. Hu, F. C. Yeh, S. H. Chuang, H. Y. Lin, M. L. Wu, C. H. Chen, and C. H. Tsai. A 0.2nJ/pixel 4K 60fps main-10 HEVC decoder with multi-format capabilities for UHD-TV applications. In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014 - 40th*, pages 195–198, Sept. 2014. doi:10.1109/ESSCIRC.2014.6942055.

[128] H. Kim, S. Cho, K. Byun, and N.-W. Eum. Multi-core based HEVC hardware decoding system. In *2014 IEEE International Conference on Multimedia and Expo Workshops (ICMEW)*, pages 1–2, July 2014. doi:10.1109/ICMEW.2014.6890626.

[129] M. Tikekar, C. T. Huang, C. Juvekar, V. Sze, and A. P. Chandrakasan. A 249-Mpixel/s HEVC video-decoder chip for 4k Ultra-HD applications. *IEEE Journal of Solid-State Circuits*, 49(1): 61–72, Jan. 2014. ISSN 0018-9200. doi:10.1109/JSSC.2013.2284362.

[130] T. M. Liu, Y. C. Chang, C. M. Wang, H. M. Lin, C. Y. Cheng, C. C. Chen, M. H. Chiu, S. J. Wang, P. Chao, M. J. Hu, F. C. Yeh, S. H. Chuang, H. Y. Lin, M. L. Wu, C. H. Chen, C. L. Ho, and C. C. Ju. Energy and area efficient hardware implementation of 4K Main-10 HEVC decoder in Ultra-HD blu-ray player and TV systems. In *2015 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, June 2015. doi:10.1109/ICME.2015.7177399.

[131] P. T. Chiang, Y. C. Ting, H. K. Chen, S. Y. Jou, I. W. Chen, H. C. Fang, and T. S. Chang. A QFHD 30-frames/s HEVC decoder design. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(4):724–735, Apr. 2016. ISSN 1051-8215. doi:10.1109/TCSVT.2015.2409019.

[132] M. Abeydeera, M. Karunaratne, G. Karunaratne, K. De Silva, and A. Pasqual. 4K real-time HEVC decoder on an FPGA. *IEEE Transactions on Circuits and Systems for Video Technology*, 26(1):236–249, Jan. 2016. ISSN 1051-8215. doi:10.1109/TCSVT.2015.2469113.

[133] NVIDIA. *NVIDIA Video Decoder (NVCUVID) Interface*, version 7.5: Sept. 2015.

[134] F. Bossen. *Common test conditions and software reference configurations*. Doc. JCTVC-L1100 of JCT-VC, Jan. 2013.

[135] L. Haglund. *The SVT High Definition Multi Format Test Set*. Sveriges Television AB (SVT), Sweden, 2006.

[136] Joint Collaborative Team on Video Coding (JCT-VC) of ISO/IEC MPEG and ITU-T VCEG. Subversion repository for the HEVC test model version HM 15.0, accessed: 2016, Oct. 01. URL `https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-15.0/`.

[137] NVIDIA. *NVIDIA® CUDA™ Profile User's Guide*, version 7.5: Sept. 2015.