GHEVC: An Efficient HEVC Decoder for Graphics Processing Units

Diego F. de Souza, *Student Member, IEEE*, Aleksandar Ilic, *Member, IEEE*, Nuno Roma, *Senior Member, IEEE*, and Leonel Sousa, *Senior Member, IEEE*

***Abstract*—The high compression efficiency that is provided by the high efficiency video coding (HEVC) standard comes at the cost of a significant increase of the computational load at the decoder. Such an increased burden is a limiting factor to accomplish real-time decoding, specially for high definition video sequences (e.g., Ultra HD 4K). In this scenario, a highly parallel HEVC decoder for the state-of-the-art graphics processor units (GPUs) is presented, i.e., GHEVC. Contrasting to our previous contributions, the data-parallel GHEVC decoder integrates the whole decompression pipeline (except for the entropy decoding), both for intra- and interframes. Furthermore, its processing efficiency was highly optimized by keeping the decompressed frames in the GPU memory for subsequent inter frame prediction. The proposed GHEVC decoder is fully compliant with the HEVC standard, where explicit synchronization points ensure the correct HEVC module execution order. Moreover, the GPU-based HEVC decoder is experimentally evaluated for different GPU devices, an extensive range of recommended HEVC configurations and video sequences, where an average frame rate of 145, 318, and 605 frames per second for Ultra HD 4K, WQXGA, and Full HD, respectively, was obtained in the Random Access configuration with the NVIDIA GeForce GTX TITAN X GPU.**

***Index Terms*—Graphics processor units (GPUs), high efficiency video coding (HEVC), parallel processing, real-time, video decoding.**

I. Introduction

When compared with previous standards, the High Efficiency Video Coding standard [1], developed by the Joint Collaborative Team on Video Coding, has shown to reduce by half the bitrate required to compress a video sequence with the same visual quality [2]. However, to achieve this HEVC compression efficiency, the computational load is increased in both the encoder and the decoder [3]. In order to achieve the best compromise between distortion, compression rate and computational complexity, a set of HEVC features and parameters, e.g., motion vectors, block partitioning and quantization step, can be carefully selected and configured at the encoder side. On the other hand, a HEVC decoder has to decompress any compliant bitstream, for a set of profiles, levels and tiers, regardless of the involved computational complexity in the decoding procedure. This requirement often leads to arduous challenges, when implementing real-time HEVC decoders.

In order to achieve high compression rates, 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, 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. Moreover, besides the ability to comply with these computationally complex and highly data dependent operations, a real-time decoder has also to fulfill the strict frame rate requirements, which is difficult to be achieved when processing high resolution frames.

In this scenario, to provide real-time capabilities by accelerating the HEVC decoder procedures, several implementations have been proposed for: Field-Programmable Gate Array(FPGAs) [4], multicore Central Processing Unit(CPUs) [5], Digital Signal Processor(DSPs) [6] and Graphics Processing Unit(GPUs) [7]. Among those implementations, the multicore CPU and GPU are the most common setup for nowadays heterogeneous systems, e.g., desktops, laptops and smartphones, where the GPU is optimized for highly data parallel applications. However, to fully exploit the GPU architecture, the targeted application usually must be redesigned, in order to maximize the degree of parallelism and to take advantage of the GPU memory hierarchy and high execution concurrency.

Herein, a comprehensive GPU-based HEVC decoder is proposed, that allows both intra and inter frame processing, which are only partially or individually tackled in our previous contributions [7]–[9]. In this way, to offer a complete GPU-based solution, the contributions that are presented in this paper can be summarized as follows:

1) A comprehensive redesign of all HEVC decoder procedures in order to decode a video sequence in the GPU device, which means to maximize the parallelism level, optimize the memory accesses and increase the instruction throughput.

Manuscript received March 7, 2016; revised July 19, 2016; accepted October 18, 2016. Date of publication November 3, 2016; date of current version February 14, 2017. This work was supported by the National Funds through Fundação para a Ciência e a Tecnologia (FCT) under Project PTDC/EEI-ELC/3152/2012 and Project UID/CEC/50021/2013. The work of D. F. de Souza was supported by the FCT under the Ph.D. scholarship SFRH/BD/76285/2011. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Yonggang Wen.

The authors are with the INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisboa 1000-029, Portugal (e-mail: difs@sips.inesc-id.pt; ilic@sips.inesc-id.pt; nuno.roma@inesc-id.pt; las@inesc-id.pt).

Digital Object Identifier 10.1109/TMM.2016.2625261

TABLE I
LIST OF ACRONYMS

Acronym	Full name
BF	Bypass Flag
BS	Boundary filtering Strength
CBF	Coded Block Flag
CPU	Central Processing Unit
CTU	Coding Tree Unit
CU	Coding Unit
CUDA	Compute Unified Device Architecture
DBF	Deblocking Filter
DIT	De-quantization and Inverse Transform
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
FPS	frames per second
GHEVC	GPU-based HEVC
GPU	Graphics Processing Unit
HEVC	High Efficiency Video Coding
IP	Intra Prediction
JCT-VC	Joint Collaborative Team on Video Coding
Mbps	megabit per second
MC	Motion Compensation
PB	Prediction Block
PU	Prediction Unit
QP	Quantization Parameter
SAO	Sample Adaptive Offset
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SM	Streaming Multiprocessor
TB	Transform Block
TBF	Transquant Bypass Flag
ThB	Thread Block
TSF	Transform Skip Flag
TU	Transform Unit
WPP	Wavefront Parallel Processing

2) A unified design of GHEVC features, which reinforces data sharing among different HEVC procedures by taking advantage of the GPU memory hierarchy.
3) A frame-level GPU parallel processing, where different parts of the frame are processed in parallel, while ensuring the HEVC standard compliancy.

According to the conducted experimental evaluation in the state-of-the-art GPU devices, the GHEVC decoder can handle *Ultra HD 4K* video sequences by delivering up to 200 frames per second for low bitrate and around 80 FPS for high bitrate in the *Random Access* configuration.

This paper is organized as follows. The background and related work are presented in the Section II. The GHEVC decoder designs are described in Section III, while modules integration are comprehensively explained in Section IV. The profiling and evaluation of the GHEVC decoder are discussed in Section V. Finally, Section VI addresses the most important conclusions and future work.

To facilitate the readability of the paper, Table I presents the list of acronyms used in the rest of the paper.

II. Background and Related Work

In HEVC, a video frame is decoded from the received bitstream in data elements corresponding to square pixel blocks. These pixel blocks are denoted as Coding Tree Unit(CTUs) [10], whose size ($N \times N$) is decoded from the received bitstream.

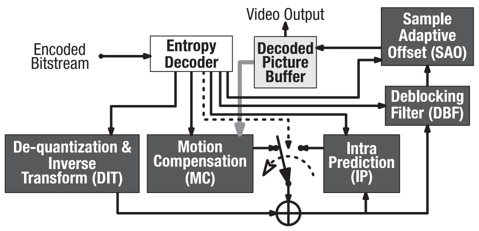


Fig. 1. High-level HEVC decoder framework.

Possible values for N are 64, 32 and 16 pixels [1]. Each CTU is further split in square blocks, named Coding Unit(CUs), by following a quadtree structure [10].

A. HEVC CU Decoding

Each CU encloses a Prediction Unit and a Transform Unit, responsible for generating the prediction pixel block and corresponding residual data, respectively. The prediction pixel block can either be obtained by using data from the same frame (intra prediction) or from previous decoded frames (inter prediction). In a HEVC decoder, each CTU and the corresponding Coding Units (CUs) are processed by the following modules:

1) *entropy decoder:* decodes the input bitstream and collects the required data to decompress the video sequence.
2) *de-quantization and inverse transform:* recovers the residual data from each TU of the frame.
3) *motion compensation:* obtains the PU inter prediction, by considering the previously decoded frames in the Decoded Picture Buffer as reference frames.
4) *intra prediction:* executes the PU intra prediction.
5) *deblocking filter:* reduces the block artifacts from the block-based hybrid video coding.
6) *sample adaptive offset:* improves the overall image quality, by reducing the CTU sample distortion.

A general framework of a HEVC decoding structure, together with the corresponding module integration, is presented in Fig. 1. First, the *Encoded Bitstream* is decoded by the *Entropy Decoder* module in order to acquire the compressed data from its input bitstream and to distribute the decoded data to the other modules. Then, the pixel blocks are reconstructed, by adding the residual block from the DIT module and the contents of the prediction block, computed either in the IP or in the MC modules. When the reconstructed frame is obtained, the in-loop filters (i.e., DBF and SAO) are applied, in order to obtain the final video frame, which is later stored in the *Decoded Picture Buffer*. Finally, the *Video Output* is obtained from the *Decoded Picture Buffer*, which is also used for storing the references frames for the MC module.

In the herein proposed GHEVC decoder, all the modules presented with dark gray (see Fig. 1) are executed by the GPU, while the *Entropy Decoder* is performed by the CPU due to its sequential and irregular nature.

B. Related Work

When considering CPU-based decoders, the HEVC reference software HM [11] is still the most used decoder for research purposes. However, it is not optimized for practical applications neither it targets real-time performance. As a consequence, the open-source OpenHEVC [12] decoder, heavily optimized for Single Instruction, Multiple Data vectorization, is usually regarded as a more suitable benchmark and it will be herein adopted for the performance evaluation.

In [5], SIMD instructions were also exploited to speed up the HEVC decoder, where 40-75 FPS were obtained for *Ultra HD 4K* HEVC videos on an Intel i7-2600 3.4GHz quad-core processor with four decoding threads. In the same direction, Chi et al., evaluated the performance of several recent SIMD ISAs for all HEVC decoder modules, where 133 FPS was achieved when decoding *Full-HD* video sequences using only one CPU core [13].

In what concerns hardware implementations of the HEVC decoder, the authors in [14] applied a set of optimizations to designing and implement an application specific integrated circuit in 40nm CMOS technology. The developed architecture is able to decode 30 FPS of *Ultra HD 4K* video sequences at 200 MHz. In [15], *Ultra HD 4K* video sequences can be decoded at 60 FPS in a 28 nm CMOS chip running at 350 MHz. Regarding FPGA implementations, in [16] it is presented a decoder architecture that is able to decode an *Ultra HD 4 K* video at 30 FPS in a Xilinx Zynq 7045 FPGA, running at 150 MHz.

When considering GPU devices to accelerate the video codec, most of the presented works in the literature tackle only the encoder side, due to its higher computational load and optimization challenges (e.g., how to efficiently cope with a Lagrangian cost function used in the mode decision or motion estimation). Some examples are presented in [17] for the H.264/MPEG-4 AVC and in [18] for the HEVC.

Besides these contributions at the encoder side, our previous research relied on GPU devices to accelerate some HEVC decoder modules. In accordance, the main differences with the new contributions that are herein presented are the following:

1) *DIT:* In [19], multiple GPU kernels are assigned to implement the DIT module, one for each TU size. Moreover, the DIT implementations presented in [7] and [8] do not support 4×4 inverse transform of inter predicted CUs, since only intra frames (i.e., intra predicted blocks) are considered. In contrast, the herein proposed DIT kernel is an unified single kernel implementation, where both inter predicted CUs and all TU sizes are supported.
2) *MC:* In [9], 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 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 herein proposed decoder allows a significant reduction of the superfluous memory transfers between CPU and GPU of the reference frames and predicted frame. Additionally, a complete GPU-based *Decoded Picture Buffer* is provided.
3) *IP:* In [7] and [8], the decoding of intra predicted CUs inside an inter frame is not implemented in the GPU, since both works only support intra frames. In contrast, the GHEVC decoder already supports intra predicted blocks in inter frames. This capability is achieved by coupling the IP GPU kernel after the MC GPU kernel execution and by explicitly considering if the 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 has been improved, in order to ensure a better load balancing across the available Streaming Multiprocessor(SMs).
4) *DBF:* In [20], the filtering decisions are calculated in the CPU side and subsequently sent to the GPU. In [8], 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 and 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. In this way, the proposed DBF kernel provides the full support for both inter and intra predicted blocks (while [8] only supports the intra blocks).

Moreover, since only intra frames are considered in [8], the execution order of the kernels from different parts of the frame is imposed by the data dependencies intrinsic to the IP module, when processing different parts of the frame in parallel. Nonetheless, explicit synchronization points are proposed herein, in order to guarantee the correct execution order of the in-loop filters for different parts of the frame, since both intra and inter predictions are considered.

When considering other 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, most hardware-based HEVC decoders in current GPU devices are only available on 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 [21]). In this scenario, the dedicated hardware is usually accessed through a specific API (such as the Microsoft DirectX Video Acceleration[1]) and implemented in the wrapping software (e.g., LAV Filters[2]).

Regarding software decoding on Graphics Processing Units (GPUs), OpenCL has been also used to provide HEVC decoding capabilities in several commercial decoders. For example, the Ittiam's i265[3] 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 Stron-

[1][Online] Available: https://msdn.microsoft.com/en-us/library/aa965263.aspx

[2][Online] Available: https://github.com/Nevcairiel/LAVFilters

[3][Online] Available: http://www.ittiam.com/products/software-ips/video/h265-hevc/

gene OpenCL H.265/HEVC Decoder for Windows.[4] CyberLink PowerDVD also provides OpenCL-based HEVC decoder capabilities.[5] However, a direct comparison with these commercial applications is difficult to provide, 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. Hence, to the best of our knowledge, the herein presented work is the first academic GPU-based solution to implement the complete HEVC decompression in the GPU. In fact, most state-of-the-art approaches only deal with a GPU parallelization of a single HEVC module, while the remaining modules are usually not taken into consideration, e.g., the GPU-based DBF in [22] and [23], and the GPU-based DIT in [24]. In the following sections, the proposed implementations for each GHEVC module are briefly presented, as well as the proposed GHEVC decoder.

III. HEVC Decoder Modules: GPU Parallelization

To fully exploit the GPU capabilities, the commonly used Compute Unified Device Architecture [25] 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 requirements should be specifically considered [25], this is:

1) *fine-grain parallelism:* each HEVC module should be implemented in a way that it exposes as much data parallelism as possible, which allows a large number of simultaneously active threads.
2) *memory optimizations:* the data accesses for 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.
3) *instruction throughput:* the GPU execution is organized in groups of 32 parallel threads called warps, by following the parallel execution model Single Instruction, Multiple Thread, where all threads in a warp perform the same operation from the GPU code (kernel). In this case, a divergence control flow instructions between threads in a warp (warp divergence) should be avoided, since the different executions paths have to be serialized, thus decreasing the overall performance of the GPU kernel.

Moreover, the warps are grouped in several Thread Block(ThBs) and the proposed algorithms maximize the number of active Thread Blocks (ThBs) in order to achieve the highest performance.

[4][Online] Available: http://www.strongene.com/en/downloads/downloadCenter.jsp

[5][Online] Available: http://www.cyberlink.com/products/powerdvd-ultra/features_en_EU.html

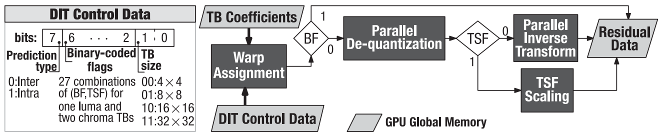


Fig. 2. Control data and flowchart of the GPU-based HEVC DIT module.

As follows, each GHEVC decoder module is briefly presented to explain how the three main requirements, denoted as *Fine-grain Parallelism*, *Memory Optimizations* and *Instruction Throughput*, are fulfilled.

A. Dequantization and Inverse Transform

To decompress the residual data of each Transform Block from luma and two chroma components in a TU, the DIT module starts by acquiring a set of input data from the *Entropy Decoder*: *i)* the TB coefficients and *ii)* the transform flags, i.e., Coded Block Flag, Transform Skip Flag and Transquant Bypass Flag.

1) Fine-Grain Parallelism: The proposed GPU-based DIT module has high degree of fine-grain parallelism, since all the Transform Blocks (TBs) in the frame can be processed in parallel. However, since the frame TB partitioning is unknown a priori, each warp in a ThB is assigned to a 4×4 block of the CTU. Then, the warps are assigned according to the respective TB size, e.g., eight warps perform the DIT for a 32×32 TB, while only one warp processes a 4×4 TB, as in [8].

2) Memory Optimizations: The required data to perform the DIT module is packed in such a way that only one memory transaction to the high latency GPU global memory is necessary to obtain the prediction type, TB size and transform flags. For this purpose, the CBF and TBF are merged in a new flag denoted as Bypass Flag [19], and the TB coefficients are stored as *Residual Data* whenever the BF is set. In this case, there are three possible flag combinations per component (luma or chroma), i.e., (BF,TSF)$\in\{(0,0);(0,1);(1,\cdot)\}$, which leads to 27 flag combinations per TU.

As it can be observed in Fig. 2, the *DIT Control Data* contains all the information required to perform the DIT, which are packed in a single 1-byte word per 4×4 luma pixel block of the frame. The *DIT Control Data* includes:

1) *TB size (bits 0 and 1):* indicates one of the four possible TB sizes, i.e., 4×4, 8×8, 16×16 and 32×32.
2) *binary-coded flags (bits 2 to 6):* represent 27 possible flags combinations of BF and TSF for all TBs in a TU, which can be recovered with bitwise operations.
3) *prediction type (bit 7):* signals if the TB belongs to an intra or an inter predicted CU, in order to select which *Transform Coefficient Array* is used for 4×4 TBs.

Since the *Transform Coefficient Arrays* are constants, they are stored in the GPU texture memory to take advantage of the 2D GPU texture cache in the matrix multiplication of the inverse transform procedure [8]. Moreover, the intermediate values of the DIT procedure are stored in the low latency GPU shared memory, which provides faster accesses and single memory transactions to access block rows or columns, since there are no bank conflicts. Finally, the *DIT Control Data* for all 4×4

blocks of the frame, the TB coefficients and the residual data output are stored in the GPU global memory.

3) Instruction Throughput: In the proposed GPU DIT kernel, the warp divergence is avoided since each warp performs the DIT procedure for a TU or a partition of the TU. As a result, all threads in a warp always follow the same execution path.

The flowchart of the proposed DIT module is presented in Fig. 2. At the beginning of DIT processing, in *Warp Assignment*, the warps are designated to a specific TU (or a part of the TU), according to the *TB size*. If the BF is unset, a *Parallel De-quantization* procedure is performed on the *TB Coefficients* (one coefficient per GPU thread), where the de-quantized coefficients are stored in the GPU shared memory for subsequently accesses.

Afterwards, if the TSF is set, the inverse transform is skipped and the final scaling (i.e. *TSF Scaling*) is performed before the de-quantized coefficients are considered as *Residual Data*. If not (i.e. TSF=0), the *Parallel Inverse Transform* is performed within two parallel matrix multiplications with multiple warps that access the *Transform Coefficient Array* in the GPU texture memory and the TB in the GPU shared memory without bank conflicts. Moreover, synchronization points are used in the *Parallel Inverse Transform* to ensure the memory coherency on the GPU shared memory. The final result is stored in the GPU global memory as a residual data block from each TB, as also referred in [8].

B. Motion Compensation

Each PU is composed of Prediction Block(PBs) of luma and chroma components in both intra and inter prediction modes. In an inter predicted CU, the PB partition belongs to one of four symmetric or four asymmetric partitioning modes [26]. Herein, 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.

1) Fine-Grain Parallelism: Although the prediction of each inter PB can be performed in parallel, the shape and size of smaller Prediction Blocks (PBs) are limiting factors for the GPU performance, due to the irregular memory accesses on the reference frames. Nevertheless, the warp can perform 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 pixel width. This requirement guarantees 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. Since the worst case scenario is for the chroma component with chroma subsampling 4:2:0, in the proposed MC module the warp operates at a luma block of $64 \times N$, which leads to a $32 \times N/2$ chroma block.

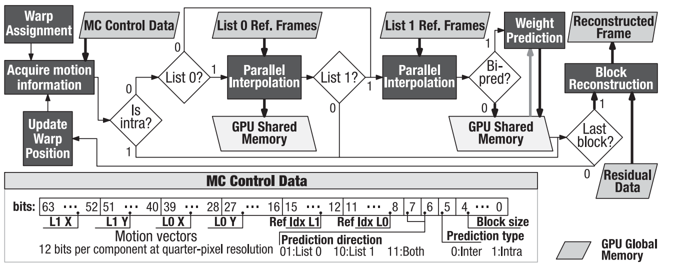


Fig. 3. Flowchart and control data of the GHEVC MC module.

Due to the lower latency memory accesses in comparison with the GPU global memory, the GPU shared memory is also used as temporary storage for the interpolation procedure. Herein, a pixel block from the reference frame is fetched from the GPU global memory to the shared memory. However, the GPU shared memory is a very limited resource, 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 $64 \times N$ 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: parallelism degree, amount of requested shared memory, number of active warps and global memory bandwidth. In this way, eight warps are assigned per ThB, which performs the GPU-based MC module in a 64×64 pixel block of the frame. Furthermore, each warp performs 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 perform the inter prediction and reconstruction of a 64×8 sub-part of PB.

2) Memory Optimizations: The motion data, to perform the proposed inter prediction of a PU, is packed into a 64-bit word, as presented in Fig. 3 (*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 smaller block (i.e., two 4×8 or two 8×4 blocks). Moreover, the *MC Control Data* for the whole frame is stored in the GPU global memory, which can be retrieved with a single memory transaction to perform the motion compensation of a single PU. In this way, the 64-bit word of the *MC Control Data* is structured as:

1) *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.
2) *prediction type (bit 5):* signals if the CU is intra (1) or inter predicted (0).
3) *prediction direction (bits 6 and 7):* indicates if List 0 (bit 6) and List 1 (bit 7) reference frames are used.
4) *ref Idx L0 (bits 8 to 11):* index of the chosen reference frame from a set of 16 possible values from List 0 (L0).
5) *ref Idx L1 (bits 12 to 15):* index of the chosen reference frame from a set of 16 possible values from List 1 (L1).
6) *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).

Furthermore, the whole frame is decompressed in the GPU, which means that the reference frames are already stored in the GPU global memory. In this case, the List 0 and List 1 are arrays of pointers (to the reference frames) stored in the GPU constant memory, which are updated for each new frame. The constant memory is also used to store the interpolation filter coefficients. Finally, the residual data is retrieved from the GPU global memory (the output of the GPU-based DIT module).

3) Instruction Throughput: Fig. 3 presents a simplified flowchart of the overall inter prediction procedure executed 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 Fig. 3). Otherwise, the block is inter predicted and bit 6 signals if the reference frame belongs to List 0 (see "*List 0?*" decision in Fig. 3). If bit 6 is set, the *Parallel Interpolation* [9] 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 Fig. 3) is inspected in order 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 (see *Update Warp Position* in Fig. 3).

When all blocks of the 64×8 pixel block assigned to a single warp are predicted (i.e., "*Last block?*" decision returns one), the reconstructed block is generated 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 Fig. 3). It is important to notice 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 instruction, to improve the performance.

Moreover, the overall procedure avoids warp divergence, since all threads in a warp always follow the same execution path in both prediction and reconstruction steps, where the GPU shared memory accesses were carefully designed in order to avoid bank conflicts [9]. The final reconstructed 64×8 luma pixel block is stored in the GPU global memory, as part of the *Reconstructed Frame* for the subsequent GHEVC decoder modules.

C. Intra Prediction

Like in the MC module, the proposed IP module performs the 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 [27]. In this way, 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.

1) Coarse-Grain Parallelism: To perform the intra prediction, the pixels from the reconstructed neighboring blocks are used as reference samples. These intrinsic data dependencies between intra predicted pixel blocks limit the level of parallelism within the IP module. Nevertheless, a coarse-grain parallelism can be obtained by executing the intra prediction in a wavefront approach for the whole frame [7], instead of processing a single CTU row at a time. In this way, the intra prediction of a frame pixel block can be carried out when the needed 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 progress with 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 [7]. 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 adopted, the value of $N = 8$, since it provides the best compromise between the granularity of the wavefront processing (parallelism degree) and the amount of GPU global memory accesses to check dependencies. When compared to [8], one of the contributions of the herein proposed IP kernel is a better load balancing across the Streaming Multiprocessors (SMs), where the neighboring 8 pixel rows of the frame are processed by different thread blocks. In this case, the workload of the wavefront approach is efficiently distributed according to the existing GPU resources.

2) Memory Optimizations: Since the intra prediction is performed at a TB level, the necessary data to perform the IP module are: *i*) the TB size and prediction type, which are pro-

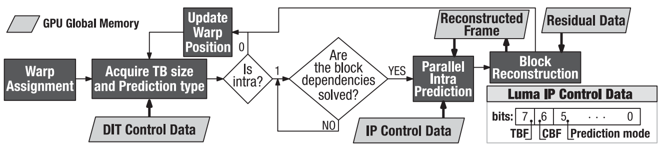


Fig. 4. Flowchart and control data of the GHEVC IP module.

vided by the *DIT Control Data*; *ii*) the intra prediction mode; and *iii*) the reconstructed frame. In Fig. 4, the luma *IP Control Data* is presented, which is contained in a single byte structure, as follows:

1) *prediction mode (bits 0 to 5):* encodes 35 intra prediction modes (Planar, DC and 33 Angular modes) and an extra mode for the PCM.
2) *CBF and TBF (bit 6 and 7):* reserved for the 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 [7] and [8], 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 [25].

3) Instruction Throughput: The flowchart of the IP module is presented in Fig. 4, where each warp is firstly assigned by the *Warp Assignment* to its own 8 pixel row of the frame. Starting from the first pixel block (on the left of the 8 pixel row) until the end of the pixel row, the following procedure 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 Fig. 4) to obtain the *Prediction type* (bit 7) and *TB size* (bits 0 and 1). Bit 7 is checked in the "*Is Intra?*" decision (see Fig. 4), 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 Fig. 4). 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 needed neighboring blocks. Then, the warp "positions" of the corresponding neighboring blocks are verified, in order to check if the dependencies are solved (see Fig. 4). 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 accesses. 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 Fig. 4) is performed, where each thread is responsible for one or more pixels of the block, as presented in [7].

In the *Block Reconstruction* procedure depicted in Fig. 4, the predicted block is added to the residual data, where the final reconstructed block is stored in the GPU global memory to be used as reference for the next neighboring blocks. 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.

D. Deblocking Filter

The HEVC deblocking filter is applied for each PU or TU boundaries, which belong to an 8×8 grid of the frame, according to specific conditions defined in the standard [28]. In the first condition, the edge is filtered only if the Boundary filtering Strength is higher than 0, which is calculated according to the decoded data of the blocks on each side of the boundary. The possible BS values are equal to: *i*) 2, if at least one of the blocks at the boundary is intra predicted; *ii*) 1, depending on specific conditions of the motion information of both blocks; or *iii*) 0, otherwise [1], [28].

1) Fine-Grain Parallelism: Consecutive boundaries in the DBF module can be processed in parallel, since only up to 3 pixels are filtered and up to 4 pixels are read on each boundary side, where the vertical boundaries are filtered before the horizontal ones. To extract the fine-grain parallelism of the DBF module, as referred in [20], the independent 8×8 pixel blocks of the frame are processed in parallel, by shifting the pixel block by four pixels in the horizontal and vertical component in comparison with the 8×8 grid of the frame.

These 8×8 pixels blocks are completely independent and a set of eight blocks are assigned for each warp. In this way, similarly to the MC module, each warp operates on a 64×8 pixel block of the frame, which provides the efficient memory bandwidth utilization of the GPU global memory for luma and chroma components. Moreover, two boundaries can be simultaneously processed inside the 8×8 pixel block, i.e., two vertical (8×4) or horizontal (4×8) boundaries.

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

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

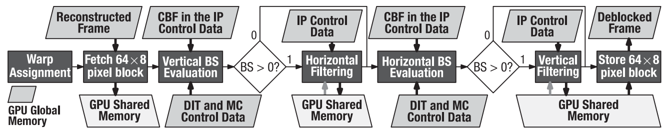


Fig. 5. Flowchart of the GHEVC DBF module (luma component).

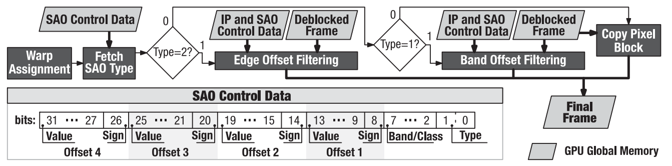


Fig. 6. Flowchart and control data of the GHEVC SAO module (luma or chroma component).

4) *motion discrepancy:* the *MC Control Data* of both boundary blocks are used to check if they have different reference frames, a 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*, where 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 GPU shared memory is used to store temporary values during the filtering, where the whole 64 $\times$ 8 pixel block is fetched from the reconstructed frame, filtered and stored back in the GPU global memory as part of the deblocked frame.

3) Instruction Throughput: A general diagram of the proposed DBF module is presented in Fig. 5. First, the warps are assigned to distinct 64 $\times$ 8 regions of the frame by the *Warp Assignment* procedure. Then, the assigned 64 $\times$ 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 process (see *Fetch 64 $\times$ 8 pixel block* in Fig. 5). The BS values of each vertical edge in the 64 $\times$ 8 pixel block are simultaneously evaluated (see *Vertical BS Evaluation* in Fig. 5), 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 [8]. 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 Fig. 5).

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 Fig. 5). If the BS value is greater than zero, the *Vertical Filtering* is executed [8]. Finally, the filtered 64 $\times$ 8 pixel block is stored in the GPU global memory as part of the *Deblocked Frame*.

E. Sample Adaptive Offset

In the SAO filtering, two procedures are specified, i.e., the SAO Edge Offset and the SAO Band Offset, in order to filter all pixels in the frame, where the parameters are defined per each component of the CTU [29]. After the SAO filtering, which reduces the pixel distortion, the final frame is obtained.

1) Fine-Grain Parallelism: In the SAO filtering procedure, all pixels of the deblocked frame can be filtered in parallel. However, in the proposed GPU-based SAO module, each warp processes a CTU pixel block, since the SAO filtering is applied in a CTU basis, as referred in [8]. This approach avoids multiple requests of the SAO parameters in the GPU global memory. Hence, 32 pixels of a CTU are filtered in parallel, for both luma and chroma components.

2) Memory Optimizations: The SAO parameters are packed into a 32-bit word for each frame component of the CTU, leading to 12 bytes per CTU, which are stored in the GPU global memory. The chosen data structure of the *SAO Control Data* is presented in Fig. 6 and it comprises the following fields:

1) *type (bit 0 and 1):* indicates if it is filtered as Edge Offset (*Type* = 2), Band Offset (*Type* = 1) or neither (*Type* = 0).
2) *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 [8].
3) *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.

3) instruction throughput: The overall procedure of the implemented SAO module (luma or chroma component) is presented in Fig. 6. 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 filtering is applied (see *Fetch SAO Type* in Fig. 6). 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 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 component part of the frame is stored in the GPU global memory (see *Final Frame* in Fig. 6). Finally, if *Type* is equal to 0, the corresponding component part of the *Deblocked Frame* is directly copied to the *Final Frame* by the *Copy Pixel Block* procedure.

In both filtering procedures, the *IP Control Data* is used to check if the pixels belong to a PCM predicted block (when the *pcm_loop_filter_disabled_flag* is set) and if the TBF is set. In either case, the pixels from those blocks are not filtered.

IV. GPU-Based HEVC Decoder Implementation

The proposed GHEVC decoder is supported by a heterogeneous system composed by a CPU and the GPU as an accelerator. The CPU is responsible for the entropy decoder and for ensuring the correct execution order of the GPU kernels, as well as for the memory transfers to the GPU memory spaces. The GPU constant and texture memory spaces are mostly used to store information that does not change along the video se-

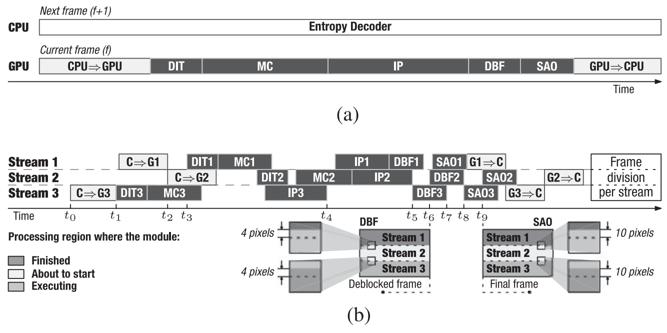


Fig. 7. Proposed processing order of an inter frame with multiple CUDA Stream configurations and frame division per stream. (a) Only one CUDA Stream. (b) Three CUDA Streams.

quence decoding process. In particular, the GPU texture memory is used to save the *Transform Coefficient Arrays* [8], whereas the constant memory is used to store:

1) *frame height and width:* the frame size is employed mainly in the thread and warp positioning of each frame;
2) *HEVC tables:* including table specification for the *intraPredAngle* and *invAngle*;
3) *HEVC filter coefficients:* the interpolation filter coefficients of the luma 8-tap and the chroma 4-tap filters;
4) *HEVC flags:* the HEVC control flags that specify the modules behavior, like *pcm_loop_filter_disabled_flag* and *strong_intra_smoothing_enabled_flag*;
5) *list 0 and 1:* the reference frames, used in the *Motion Compensation* kernel, 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 to note 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, e.g., *DIT*, *MC*, *IP* and *SAO Control Data*, are transferred to the GPU global memory before the execution of the respective GPU kernels.

A. Module Execution Order

As referred in [25], at the global CPU+GPU level, the GPU execution is organized in CUDA Streams, which represent a sequence of operations which are executed sequentially (issue-order) in the GPU. To process a given frame by using a single CUDA Stream, the proposed module execution order is presented in Fig. 7(a). Hence, it is important to notice that while the GPU is processing a given frame f, the CPU can start the entropy decoding procedure for the next frame $f + 1$, in a pipeline way. In accordance, the first command to be processed is the memory transfer operation of all the required input data to the GPU global memory (see Fig. 7(a), CPU⇒GPU).

As presented in Fig. 7(a), 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 in the GPU global memory and it is used as an 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 a "in place" algorithm. In this way, to ensure compliancy 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. The final frame is then transferred back to the CPU side (see Fig. 7(a), GPU⇒CPU).

It is worth noticing 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 frame 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 frame.

B. Concurrency Control

When multiple CUDA Streams are applied, commands of different streams (kernels and CPU⇔GPU memory transfers) may run concurrently, according to the GPU capabilities [25]. In Fig. 7(b), an example of a frame processing with three CUDA Streams is presented. Since each stream is executed independently, the frame is horizontally divided in sets of CTU rows (due to the IP module dependencies), as presented in [7].

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 Fig. 7(b), 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 occupied by the kernels that are already executing in the GPU). As illustrated in Fig. 7(b), although the *DIT1* kernel from *Stream 1* can start at t_2, it only starts at t_3, because the GPU is still occupied 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 Fig. 7(b).

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-pixel 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 Fig. 7(b), which finishes its execution at t_4.

Since the DBF module operates on 8×8 blocks, which are shifted by four pixels in the vertical and horizontal components (as explained in Section III-D), the processing region per Stream

TABLE II
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 [31] (500 frames)	CrowdRun, ParkJoy, DucksTakeOff, IntoTree and OldTownCross
Configuration	All Intra, Random Access and Low Delay
QP	22, 27, 32, 37

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 $\times$ 8 grid of the chroma frames as well [28]. 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 Fig. 7(b), this effect is observed in *DBF3*, which does not start between t_4 and t_5, until *IP2* has finished. At the bottom of Fig. 7(b), 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, where the DBF from the *Stream* i starts after the IP from the *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 Fig. 7(b), the *SAO3* kernel is put on hold from t_7 to t_8 until *DBF2* is done, where explicit synchronization points are used between the SAO from the *Stream* i and the DBF from the *Stream* $i-1$. Moreover, at the bottom of Fig. 7(b), 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. Herein, 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).

V. EXPERIMENTAL RESULTS AND EVALUATION

To evaluate the performance of the proposed GHEVC decoder, the JCT-VC recommended test conditions and configurations [30] were adopted, by considering the setup summarized in Table II. For such purpose, 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 [30], the sequences with the highest frame resolution were adopted, which includes *Class A* (2560 $\times$ 1600) and *Class B* (1920 $\times$ 1080) resolutions. A new set of video sequences (*Class S*) was also used, in order to include the *Ultra HD 4K* (3840 $\times$ 2160) frame resolution, from the SVT High Definition Multi Format Test Set [31].

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 purposes. Furthermore, the several considered video sequences were encoded by setting the Quantization Parameter value from 22 to 37 (see Table II).

To encode the input video sequences, the HM 15.0 reference software [11] was used according to [30], without any Tiles and Wavefront Parallel Processing 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) *preliminary 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$\Leftrightarrow$GPU) with multiple streams;
4) *comparison with previous work:* to present the performance improvement over previous implementations;
5) *HEVC decoding performance:* to evaluate the best performance obtained with the selected hardware.

The presented evaluation considered the whole decoding structure except the entropy decoder, which was kept at the CPU side because of its highly irregular execution pattern and weak adequacy 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 [3], [5], [13], it was decided to execute it in parallel with the the remaining decoding modules, i.e., frame $(f+1)$ was entropy decoded at the same time that the previous frame (f) was processed by the remaining decoding structure (see Section IV-A).

In what concerns the hardware platforms that were used in this experimental evaluation, six different computing setups were adopted using GPUs from two NVIDIA architectures (i.e., Maxwell and Kepler) and an Intel Core i7-6700K CPU @ 4.00GHz with four cores. The six considered GPU devices are presented in Table III, 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 represents the computed average for all tested sequences with the same frame resolution.

TABLE III
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

TABLE IV
GPU KERNEL EXECUTION TIME (IN MS/FRAME) WHEN VARYING THE NUMBER OF WARPS IN A THB

	Kernel execution time [ms/frame]			
Number of warps per ThB	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

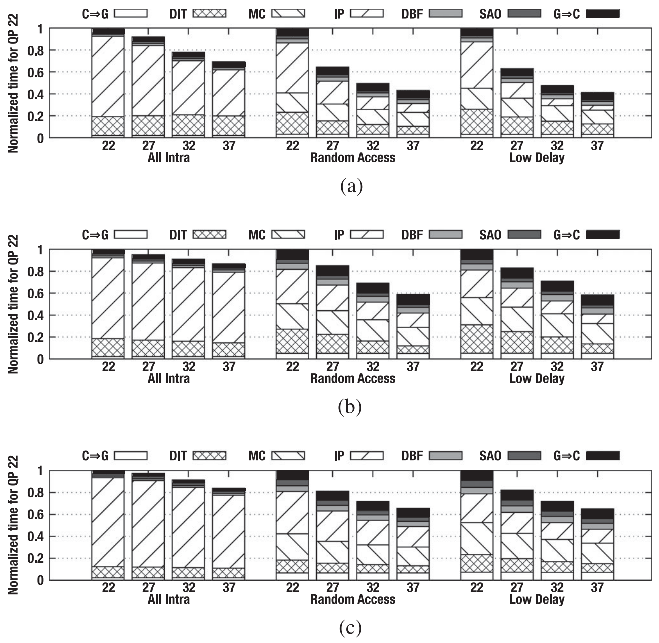


Fig. 8. Normalized frame processing time of the QP = 22 for *All Intra*, *Random Access*, and *Low Delay* configurations of a) Class S, b) Class A, and c) Class B.

A. Kernel-Level Thread Block Configuration

In this subsection, 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 approach. As explained in Section III-A, in the DIT kernel, the warps 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. In this case, a higher number of warps would force synchronization between different TBs, which are asynchronously performed in the herein proposed DIT, while a smaller number of warps would force changes in the algorithm implementation (with a possible performance loss). 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 given by 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 20 and 8, respectively.

The average execution time that is spent by each GPU kernel is presented in Table IV by considering the following configuration: *i*) one CUDA Stream; *ii*) QP value equal to 27; *iii*) all video sequences from *Class S* (3840×2160); and *iv*) *All Intra* configuration for the IP kernel and *Random Access* configuration for the remaining kernels. 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 previously presented in the Section III.

B. Preliminary Profiling Analysis

The evaluation of each individual module performance was conducted in a preliminary profiling analysis, by running one CUDA Stream in the Titan GPU. The obtained profiling results are presented in Fig. 8, by using a normalized scale to represent each individual module processing time (including memory transfers) over the overall frame processing time. For such purpose, it was considered the QP = 22 configuration, because it represents the most time consuming setup. Although the normalized memory transfers time, to and from the GPU (*C⇒G* and *G⇒C*), increases with the QP value for all classes and configurations, this overhead is always constant in a class, since it only depends on the amount of data to be processed.

Regarding the *All Intra* configuration, the IP module is the most time consuming one, as it can be observed in Fig. 8 for *Class S*, *Class A* and *Class B*. However, for all the tested classes, the IP processing time is reduced with the increase of the QP value. In fact, for higher QP values, the HEVC encoder prioritizes the frame rate over distortion, which leads to the selection of greater block sizes per CTU. Consequently, the GPU IP module on the decoder side can take advantage of these 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 that is obtained with a large wavefront.

The processing time of the remaining modules slightly varies with the considered QP values, on account to the obtained parallelism level, where the second most time demanding module in *All Intra* configuration is the DIT. Even though higher QP values imply larger TB sizes and, consequently, a smaller DIT processing time, the execution time of this kernel is mainly constrained 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, where TBs are rarely encoded as "skipped" nor "bypassed". The same is not observed for the *Random Access* and *Low Delay* configurations, where 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. Here, 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 Fig. 8). 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.

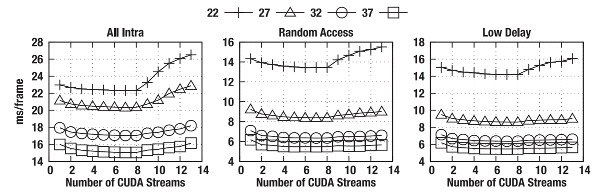
In what concerns the in-loop filters (i.e. DBF and SAO), it was observed that the overall processing time is almost constant over the tested QP values, for all configurations and classes. This is mainly due to the fact that the DBF and SAO kernels are strongly memory-bounded, which leads to an overall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

C. CUDA Streams Scalability

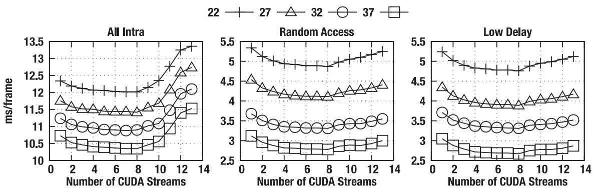
In order to evaluate the performance gains that can be obtained by overlapping the GPU kernels and memory transfers, the input bitstreams were decoded by employing up to 13 CUDA Streams in the Titan GPU. The obtained results, shown in Fig. 9, are presented as an average processing time per frame (measured in ms/frame), obtained for each class and configuration (i.e., *All Intra*, *Random Access* and *Low Delay*). Moreover, the achieved performance with each tested QP value is presented as a set of points per each configuration.

As expected, for all classes and configurations, the processing time per frame decreases when the number of CUDA Streams increases, until the minimum processing time per frame is achieved. In particular, it is possible to observe in the *All Intra* configuration that the achieved maximum performance corresponds to the usage of 8 CUDA Streams. Since the proposed GHEVC decoder bottleneck is the IP module (due to strict data dependencies between the blocks), the optimal number of CUDA Streams corresponds to the minimum processing time observed for the IP module. When more than eight CUDA Streams are employed, a consequent increase of the processing time is observed mainly due to the following three factors:

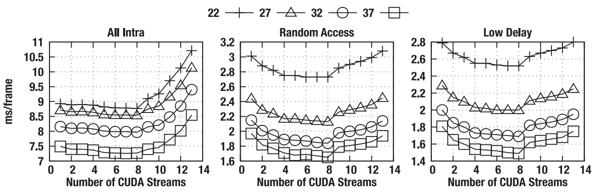
1) *used bandwidth:* when multiple streams are executed, the amount of data to be processed has to be split accordingly, in order to support independent memory transfers and kernel executions in different streams. However, a large number of small memory transfers may result in an inefficient use of the PCIe bandwidth.
2) *kernel overheads:* when launching a high number of kernels, the contribution of the time overhead associated with each kernel launch may decrease the overall performance.
3) *occupancy:* whenever the multiple kernels consume more resources than the GPU can provide, the amount of



(a)



(b)



(c)

Fig. 9. 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. (a) Class S − 3840×2160, (b) Class A − 2560 × 1600, (c) Class A − 1920 × 1080.

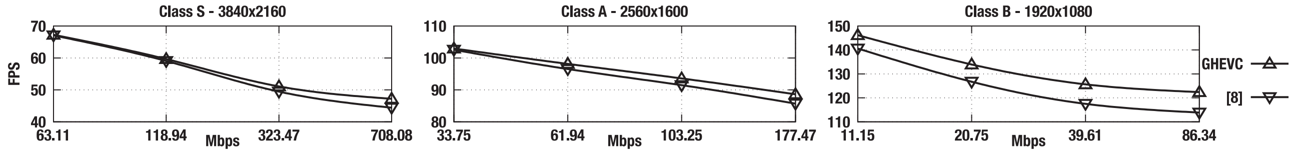


Fig. 10. Overall performance of the herein proposed GHEVC and [8] on the Titan GPU for *All Intra* configuration in *Class S*, *A*, and *B*.

simultaneously running kernels is limited, resulting in a serialized execution among the kernels.

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*, depicted in Fig. 9(b) and Fig. 9(c), respectively. For *Class S* bitstreams, the IP module is not dominant for high QP values, as it can be observed in Fig. 8(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 Fig. 8(a) and Fig. 9(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.

D. Comparison With Previous Work

Fig. 10 presents the performance improvement of the herein proposed GHEVC decoder over [8]. Since the implementation proposed in [8] 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 (FPS) measures were computed by averaging the obtained performance for all considered tested sequences for each class and QP configuration (from 22 to 37). Likewise, the obtained bitrate (corresponding to each QP) is also an average of the obtained bitrate for each sequence 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 [8] 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 Fig. 10). 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 [8], 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 [8] in the *Class B* video sequences. For *Class S* and *Class A*, a larger wavefront size (when compared to the one in *Class B*) reduces the effect of the proposed load balancing in GHEVC, since there are less idle SMs in [8] during the IP kernel execution. Nevertheless, the proposed GHEVC decoder still provides slightly higher performance than [8].

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.

E. HEVC Decoding Performance

Fig. 11 presents the experimentally obtained performance of the herein proposed GHEVC decoder, when compared with the OpenHEVC [12] CPU-based decoder executed with four threads of the i7-6700K CPU (represented as OHEVC). The OpenHEVC 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 stated in Section II, it was not possible to provide a fair and direct comparison with some existing GPU-based HEVC decoders, since they are either exploiting a dedicated GPU decoding hardware or they are closed source. The presented performance in Fig. 11 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 is an average bitrate for all video sequences within a class for each QP value.

In all presented configurations and decoders (GHEVC and OHEVC), the resulting frame rate decreases when the frame resolution is increased, on account to the greater amount of data to be processed. Even though, the proposed GHEVC decoder achieves greater 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. In fact, it can be observed that the proposed GHEVC decoder outperforms the OHEVC for the majority of 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 G980 GPU performance is slightly higher than the Titan GPU performance, in *All Intra* configuration, although the latter one owns 50% more CUDA cores than the G980 GPU. 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 16 SMs (see Table III). 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

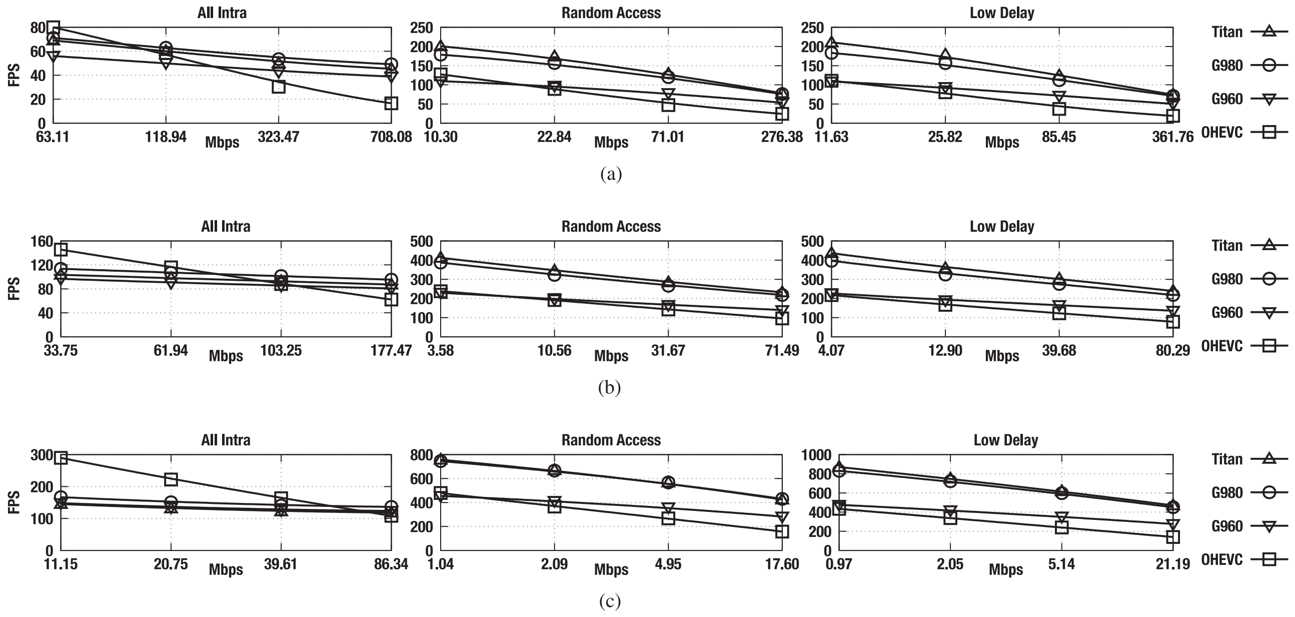


Fig. 11. Evaluation of the GHEVC decoder performance using NVIDIA Maxwell GPUs over the OpenHEVC decoder (running in the CPU).

SMs to perform 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 benefits. First, the overall utilization of cores and SMs is still limited by the amount of intrinsic IP parallelism offered in the wavefront (i.e., inherent data dependencies, as referred in Section III). 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 Fig. 11, where all three GPUs achieve a very similar performance level. However, as soon as the share of the IP in the total execution time is decreased (i.e., when the share of data-parallel MC is increased in the *Random Access* and *Low Delay* configurations – see Fig. 8), the benefits of increased number of SMs are more observable, where the best performance is achieved on Titan, followed by G980 and G960. It is worth noting that the achieved performance 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 V. In this evaluation, six GPU devices were used that belong to two NVIDIA architectures (i.e., Maxwell and Kepler), from high performance to low-end GPUs (see Table III). Table V also presents the average power consumption (in Watts) for each GPU, class and configuration.

As expected, for all GPU devices, the attained frame rate 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 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 *All Intra* configuration, the *SteamLocomotive* and the *Kimono* bitstreams provide the highest performance for *Class A* and *Class B*, since they have the largest amount of bigger PU partitions among the video bitstreams within their 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 51.0 to 61.9 FPS).

When considering the *Random Access* and the *Low Delay* configurations, it is observed that the amount of intra predicted blocks in inter 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, by consequence, are responsible for the lower performance in *Class S*, *Class A* and *Class B*, respectively, for any GPU.

In what concerns the power consumption, the obtained measures (using NVIDIA nvprof) within a single class only slightly vary for the same GPU across different configurations (see Table V). 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 energy efficiency.

In general, from the performance point of view, the proposed GHEVC decoder running on NVIDIA Maxwell GPUs outperforms its execution on Kepler devices. Even the decoding

TABLE V
AVERAGE PERFORMANCE (FPS) OBTAINED PER TESTED SEQUENCE WITH THE PROPOSED GHEVC DECODER

		All Intra						Random Access						Low Delay					
	Sequence	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	–

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*.

VI. CONCLUSION AND FUTURE WORK

An efficient GPU-based HEVC decoder, exploiting the massively parallel processing capabilities of current state-of-the-art GPU accelerators, was proposed in this paper. The presented decoder executes the whole decoding pipeline at the GPU, except for the entropy decoder module, which is kept at 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 in a GPU kernel. Moreover, all the deblocking filter decisions are entirely performed at the GPU side, by manipulating the already existing data. Furthermore, to take the maximum advantage of CUDA Streams, each frame is horizontally divided, where the set of regions that is processed by each stream is cautiously updated in order to ensure the compliancy with the HEVC standard.

A comprehensive profiling of all the GHEVC decoder modules identified the current design bottleneck, which, as expected, is the most sequential module, the *Intra Prediction*. An evaluation of the overlap between the GPU executions and the memory transfers provided an insightful knowledge on how the proposed decoder behaves with CUDA Streams. Finally, the optimized GHEVC decoder was extensively evaluated, where eight CUDA Streams assure the best overall performance. When comparing with the open-source OpenHEVC decoder (with four CPU threads), the proposed GHEVC decoder shows significant improvements in most application scenarios, by providing an average frame rate of 145, 318 and 605 frames per second for *Ultra HD 4K*, *WQXGA* and *Full HD*, respectively, in the *Random Access* configuration.

In the future, an optimized SIMD *Entropy Decoder* for CPU is expected to be integrated in the proposed GHEVC decoder, which can handle high-level parallelization techniques (i.e., Tiles and WPP). Furthermore, a synchronization scheme has to be developed, in order to conciliate multiple CPU threads working in parallel with multiple CUDA Streams.

REFERENCES

[1] 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.

[2] 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 Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1669–1684, Dec. 2012.

[3] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "HEVC complexity and implementation analysis," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1685–1696, Dec. 2012.

[4] D. Engelhardt, J. Möller, J. Hahlbeck, and B. Stabernack, "FPGA implementation of a full HD real-time HEVC main profile decoder," *IEEE Trans. Consum. Electron.*, vol. 60, no. 3, pp. 476–484, Aug. 2014.

[5] Y. Duan, J. Sun, L. Yan, K. Chen, and Z. Guo, "Novel efficient HEVC decoding solution on general-purpose processors," *IEEE Trans. Multimedia*, vol. 16, no. 7, pp. 1915–1928, Nov. 2014.

[6] 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 Trans. Consum. Electron.*, vol. 61, no. 2, pp. 236–244, May 2015.

[7] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa, "Towards GPU HEVC intra decoding: Seizing fine-grain parallelism," in *Proc. IEEE Int. Conf. Multimedia Expo*, Jun. 2015, pp. 1–6.

[8] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa, "GPU-assisted HEVC intra decoder," *J. Real-Time Image Process.*, vol. 12, no. 2, pp. 531–547, 2016.

[9] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa, "GPU acceleration of the HEVC decoder inter prediction module," in *Proc. IEEE Global Conf. Signal Inf. Process.*, Dec. 2015, pp. 1245–1249.

[10] G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[11] JCT-VC. (2014). *Subversion repository for the HEVC test model version HM 15.0.* [Online]. Available: https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-15.0/

[12] OpenHEVC. *Open source HEVC decoder (OpenHEVC)*, 2016. [Online]. Available: https://github.com/OpenHEVC/openHEVC

[13] C. C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink, and T. Schierl, "SIMD acceleration for HEVC decoding," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 5, pp. 841–855, May 2015.

[14] 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 J. Solid-State Circuits*, vol. 49, no. 1, pp. 61–72, Jan. 2014.

[15] T. M. Liu *et al.*, "Energy and area efficient hardware implementation of 4K Main-10 HEVC decoder in ultra-HD blu-ray player and TV systems," in *Proc. IEEE Int. Conf. Multimedia Expo*, Jun. 2015, pp. 1–6.

[16] M. Abeydeera, M. Karunaratne, G. Karunaratne, K. D. Silva, and A. Pasqual, "4K real-time HEVC decoder on an FPGA," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 26, no. 1, pp. 236–249, Jan. 2016.

[17] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa, "Dynamic load balancing for real-time video encoding on heterogeneous CPU+GPU systems," *IEEE Trans. Multimedia*, vol. 16, no. 1, pp. 108–121, Jan. 2014.

[18] W. Xiao, B. Li, J. Xu, G. Shi, and F. Wu, "HEVC encoding optimization using multicore CPUs and GPUs," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 11, pp. 1830–1843, Nov. 2015.

[19] D. F. de Souza, N. Roma, and L. Sousa, "OpenCL parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms," in *Proc. 22nd Eur. Signal Process. Conf.*, Sep. 2014, pp. 755–759.

[20] D. F. de Souza, N. Roma, and L. Sousa, "Cooperative CPU+GPU deblocking filter parallelization for high performance HEVC video codecs," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, May 2014, pp. 4993–4997.

[21] NVIDIA Video Decoder (NVDEC) Interface, NVIDIA Corp., Holmdel, NJ, USA, 2016.

[22] A. F. Eldeken, R. M. Dansereau, M. M. Fouad, and G. I. Salama, "High throughput parallel scheme for HEVC deblocking filter," in *Proc. IEEE Int. Conf. Image Process.*, Sep. 2015, pp. 1538–1542.

[23] W. Jiang *et al.*, "A novel parallel deblocking filtering strategy for HEVC/H.265 based on GPU," *Concurrency Comput. Practice Experience*, vol. 28, no. 16, pp. 4264–4276, 2016, cPE-15-0134.R1. [Online]. Available: http://dx.doi.org/10.1002/cpe.3751

[24] L. P. He and S. Goto, "A high parallel way for processing IQ/IT part of HEVC decoder based on GPU," in *Proc. Int. Symp. Intell. Signal Process. Commun. Syst.*, Dec. 2014, pp. 211–215.

[25] NVIDIA Compute Unified Device Architecture (CUDA) C Programming Guide, NVIDIA Corp., Holmdel, NJ, USA, 2016, v8.0.

[26] I. K. Kim, J. Min, T. Lee, W. J. Han, and J. Park, "Block partitioning structure in the HEVC standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1697–1706, Dec. 2012.

[27] J. Lainema, F. Bossen, W. J. Han, J. Min, and K. Ugur, "Intra coding of the HEVC standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1792–1801, Dec. 2012.

[28] A. Norkin *et al.*, "HEVC deblocking filter," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1746–1754, Dec. 2012.

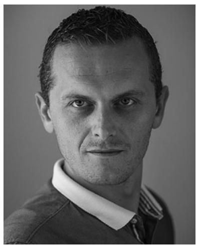
[29] C. M. Fu *et al.*, "Sample adaptive offset in the HEVC standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1755–1764, Dec. 2012.

[30] F. Bossen, *Common Test Conditions and Software Reference Configurations*, Doc. JCTVC-L1100 of JCT-VC, Jan. 2013.

[31] L. Haglund, "The SVT high definition multi format test set," Sveriges Television AB, Sweden, Tech. Rep., 2006. [Online]. Available: ftp://vqeg.its.bldrdoc.gov/HDTV/SVT_MultiFormat/SVT_MultiFormat_v10.pdf

Diego F. de Souza (S'11) received the M.Sc. degree in electrical engineering from the Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brazil, in 2010, and is currently working toward the Ph.D. degree with the Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal.

He is a Member of the SiPS Research Group at INESC-ID, under the supervision of Prof. Leonel Sousa and Prof. Nuno Roma. His current research interests include efficient GPU parallelization techniques for video coding applications, mainly based on the HEVC standard.



Aleksandar Ilic (S'09–M'12) received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon, Portugal, in 2014.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, IST, and a Senior Researcher of the Signal Processing Systems Group, Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID), Coimbra, Portugal. His research interests include high-performance and energy-efficient computing and modeling on parallel heterogeneous systems.

Nuno Roma (S'01–A'06–M'09–SM'13) received the Ph.D. degree in electrical and computer engineering from the Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon, Portugal, in 2008.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, IST, and a Senior Researcher of the Signal Processing Systems Group (SiPS), Instituto de Engenharia de Sistemas e Computadores R&D, Coimbra, Portugal. He has authored or coauthored more than 90 papers appearing in journals and international conferences, and served in the organization of several international conferences. He edited two special issues of renown international journals in the areas of energy efficient computer architectures and video encoding.His research interests include computer architectures, specialized, and dedicated structures for digital signal processing (including image and video coding and biological sequences processing), parallel processing, and high-performance computing systems.

Dr. Roma is a Senior Member of the IEEE Circuits and Systems Society and a Member of the ACM.

Leonel Sousa (M'01–SM'03) received the Ph.D. degree in electrical and computer engineering from the Instituto Superior Técnico (IST), Universidade de Lisboa (UL), Lisbon, Portugal, in 1996.

He is currently a Full Professor with UL. He is also a Senior Researcher with the R&D Instituto de Engenharia de Sistemas e Computadores, Coimbra, Portugal. He has authored or coauthored more than 200 papers appearing in journals and international conferences, and has edited four special issues of international journals. His research interests include VLSI architectures, computer architectures, parallel computing, computer arithmetic, and signal processing systems.

Prof. Sousa is a Fellow of the IET, and a Distinguished Scientist of the ACM. He has contributed to the organization of several international conferences, namely as Program Chair and as General and Topic Chair, and has given keynotes in some of them. He is currently Associate Editor of the IEEE TRANSACTIONS ON MULTIMEDIA, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, *IEEE Access, IET Electronics Letters*, Springer JRTIP, and the Editor-in-Chief of the Eurasip JES. He was the recipient of several awards, including the DASIP'13 Best Paper Award, the SAMOS'11 "Stamatis Vassiliadis" Best Paper Award, the DASIP'10 Best Poster Award, and several Honorable Mention Awards from the Universidade Técnica de Lisboa/Santander Totta (2007, 2009) and the Universidade de Lisboa/Santander (2016) for the quality and impact of his scientific publications.