Wang, B.; de Souza, D. F.; Álvarez-Mesa, M.; Chi, C. C.; Juurlink, B.;
Ilic, A.; Roma, N.; Sousa, L.

GPU Parallelization of HEVC In-Loop Filters

This is a post-peer-review, pre-copyedit version of an article published in International Journal of Parallel Programming. The final authenticated version is available online at: http://dx.doi.org/10.1007/s10766-017-0488-z.

Wang, B.; de Souza, D. F.; Álvarez-Mesa, M.; Chi, C. C.; Juurlink, B.; Ilic, A.; Roma, N.; Sousa, L. (2017). GPU Parallelization of HEVC In-Loop Filters. International Journal of Parallel Programming, 45(6), 1515–1535. https://doi.org/10.1007/s10766-017-0488-z

GPU Parallelization of HEVC In-Loop Filters

Biao Wang[1] Diego F. de Souza[2] Mauricio Alvarez-Mesa[1]
Chi Ching Chi[1] Ben Juurlink[1] Aleksandar Ilic[2]
Nuno Roma[2] Leonel Sousa[2]

Abstract In the High Efficiency Video Coding (HEVC) standard, multiple decoding modules have been designed to take advantage of parallel processing. In particular, the HEVC in-loop filters (i.e., the deblocking filter and sample adaptive offset) were conceived to be exploited by parallel architectures. However, the type of the offered parallelism mostly suits the capabilities of multi-core CPUs, thus making a real challenge to efficiently exploit massively parallel architectures such as Graphic Processing Units (GPUs), mainly due to the existing data dependencies between the HEVC decoding procedures. In accordance, this paper presents a novel strategy to increase the amount of parallelism and the resulting performance of the HEVC in-loop filters on GPU devices. For this purpose, the proposed algorithm performs the HEVC filtering at frame-level and employs intrinsic GPU vector instructions. When compared to the state-of-the-art HEVC in-loop filter implementations, the proposed approach also reduces the amount of required memory transfers, thus further boosting the performance. Experimental results show that the proposed GPU in-loop filters deliver a significant improvement in decoding performance. For example, average frame rates of 76 frames per second (FPS) and 125 FPS for Ultra HD 4K are achieved on an embedded NVIDIA GPU for All Intra and Random Access configurations, respectively.

Keywords High Efficiency Video Coding (HEVC), Graphics Processor Unit (GPU), In-loop filters, Parallelization, Decoder

Diego F. de Souza
diego.souza@inesc-id.pt

Biao Wang
biaowang@win.tu-berlin.de

Mauricio Alvarez-Mesa
mauricio.alvarezmesa@tu-berlin.de

Chi Ching Chi
chi.c.chi@tu-berlin.de

Ben Juurlink
b.juurlink@tu-berlin.de

Aleksandar Ilic
aleksandar.ilic@inesc-id.pt

Nuno Roma
nuno.roma@inesc-id.pt

Leonel Sousa
leonel.sousa@inesc-id.pt

[1] AES, Technische Universität Berlin, Einsteinufer 17, 10587 Berlin, Germany

[2] INESC-ID, IST, Universidade de Lisboa, Rua Alves Redol 9, 1000-029 Lisbon, Portugal

1 Introduction

The High Efficiency Video Coding (HEVC) standard is the state of the art in video cod-ing technology. When compared to H.264/MPEG-4 AVC, it reduces the bit rate by half without compromising the subjective visual quality [15]. This high coding efficiency, however, comes at the cost of a substantial increase in the computational load [2]. For high resolution videos, the increased workload usually makes real-time decoding very challenging, not only for embedded systems but also for desktop environments.

Fortunately, the HEVC standard has been designed with parallelism in mind, to take the advantage of parallel architectures. Most of its decoding procedures can be parallelized in order to approach to real-time decoding. For example, the HEVC in-loop filters, the last stage of the decoding pipeline, have been designed to support parallel execution. According to a profiling of the decoder conducted on an ARM Cortex-A9 processor [2], these filters account for 19–21% of the entire decoding time, which justifies the efforts for an effective parallelization. In this paper, Graphics Processing Units (GPUs) are employed to significantly improve the decoding performance of the HEVC in-loop filters, due to their wide-spread availability in mobile systems, as well as in desktop PCs.

However, although GPUs can provide high computational power, they are mostly suitable for applications with massive parallelism. This makes mapping of video decoding applications onto GPUs very challenging, since compliance with the standard is required and GPU kernels must be carefully tuned. To deliver high performance, this usually involves appropriate thread mapping, efficient memory access patterns, sufficient GPU occupancy, etc. One of the rare attempts to parallelize the in-loop fil-ters for GPUs devices is proposed in [16], which aims at leveraging the parallelism degree to improve performance. However, the herein proposed algorithms include a set of important improvements such as increased thread utilization, reduction of global memory access, and data flow optimizations. Hence, speedups of 2.0× and 1.6× are obtained when compared to over the state of the art [16] on the NVIDIA TITAN X and the NVIDIA Tegra TK1, respectively. Even for the embedded Tegra TK1 GPU with limited resources, average frame rates of 76 frames per second (FPS) and 125

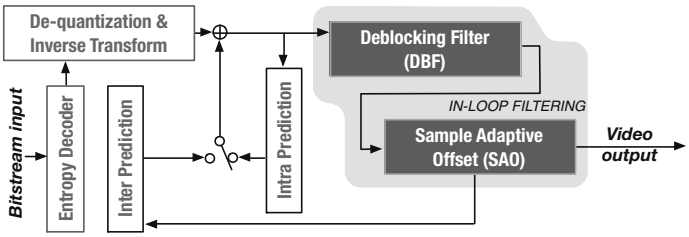

Fig. 1 A simplified HEVC decoder block diagram

FPS for Ultra HD 4K are achieved for *All Intra* and *Random Access* configurations, respectively.

Furthermore, the proposed GPU algorithm is not only compared to the state-of-the-art GPU implementation [16], but also to the state-of-the-art CPU implementation [3], in order to show the difference in the achievable performance on CPU and GPU architectures. However, it is not fair to make a direct comparison since the data granularity of the CPU and GPU algorithms is not the same. In detail, the in-loop filters are performed at the block-level on CPU in [3], while they are performed at the frame-level in the GPU-based approach. Therefore, a frame-based CPU implementation of the in-loop filters is developed to allow a fairer comparison between both devices. This paper is organized as follows. Section 2 provides a brief overview of the basic functional principles of the HEVC in-loop filters. The proposed algorithms and consequent parallel implementations are presented in Sect. 3. The obtained experimental results are discussed in Sect. 4. Section 5 revises the state-of-the-art approaches for the HEVC filtering modules and the derived conclusions are presented in Sect. 6.

2 HEVC In-Loop Filters

A generic block diagram of the HEVC decoder is depicted in Fig. 1. First, the input bitstream is decoded by the entropy decoder, in order to produce the coefficient data, as well as all other information needed to decompress the video sequence. The coefficient data is then de-quantized and inverse transformed, in order to obtain the residual data. Each block of the reconstructed frame is then computed, by adding the residual data with the predicted block from either inter or intra prediction.

The reconstructed frame is processed by the in-loop filters in order to improve the overall visual quality of the frame. In particular, to attenuate the blocking artifacts introduced by the block-based prediction and transform coding, the Deblocking Filter (DBF) is then applied at the boundaries of the reconstructed blocks. After the DBF, the mean sample distortion is further reduced with the application of the Sample Adaptive Offset (SAO) module. Finally, the output frame is produced.

In the HEVC encoding procedure, each picture is partitioned into a grid of $L \times L$ sample blocks, denoted as Coding Tree Units (CTUs), where L is dynamically selected by the encoder procedure ($L \in \{16, 32, 64\}$). The CTUs are processed in raster scan order at the decoder side. Each CTU is independently split in smaller blocks, denoted as Coding Units (CUs), according to a quadtree structure, from a maximum size of

64 × 64 samples to a minimum size of 8 × 8 samples. Additionally, each CU is further divided in Prediction Units (PUs) and Transform Units (TUs), corresponding to the prediction and to the residual blocks, respectively [18]. Inside each CTU, the CUs are decoded by following a z-scan order, as well as the PUs and the TUs within each CU.

The same frame partitioning (CTU, CU, PU and TU) is applied for each video component (i.e., luma and chroma). In particular, when the usual 4:2:0 chroma subsampling is adopted, the chroma blocks are four times smaller than the corresponding luma blocks, until the minimum size of 4 × 4 samples.

2.1 Deblocking Filter

According to the HEVC standard, the DBF is only applied to the PU and TU boundaries, which rely on a 8 × 8 sample grid for both luma and chroma. For each boundary, a Boundary filtering Strength (BS) is evaluated, according to several conditions from the neighboring blocks. The resulting BS value varies between 0 and 2, where 0 means that no deblocking filter will be applied. Whenever one of the neighboring blocks is intra-predicted, the BS value is always set to 2. Moreover, only when the BS value is two, the chroma samples are filtered [12].

On the other hand, additional conditions are verified to determine whether the DBF should be applied in luma boundaries. Each condition is verified for each set of 8 × 4 or 4 × 8 samples, corresponding to the vertical and horizontal edges, respectively (see *Boundary Types* in Fig. 2). Accordingly, a set of samples in the first and the last row (or column) are used to decide which filter is going to be applied, i.e., none, normal or strong (see black-filled samples in Fig. 2). In each side of the boundary, only up to four neighboring samples have to be considered and up to three may be modified. Taking the luma component as an example, the *strong* filtering is applied on three samples in each side of the boundary, while at most two samples may be filtered on each side of the boundary in the *normal* filtering (see *Strong Filtering* and *Normal Filtering* in Fig. 2). In contrast, the *normal* filtering is only applied on a single sample in each side of the boundary for chroma samples. Finally, the HEVC standard specifies that all

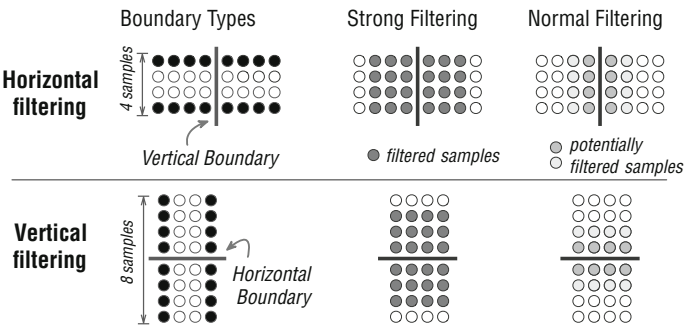

Fig. 2 HEVC deblocking filter boundary types. Filtering decisions are made based on the sample lines or columns dark-filled

vertical edges from the frame are processed by the DBF before the filtering procedure of the horizontal edges [10].

2.2 Sample Adaptive Offset

The reconstructed samples are processed by the SAO module after being filtered by the DBF module, as depicted in Fig. 1. In SAO filtering, the deblocked samples are subsequently modified by adding an offset value whose magnitude depends on a set of SAO parameters: *i*) *Type*; *ii*) four *Offset Values*; and *iii*) *Band Position* or *Edge Class*. These SAO parameters are encoded in the bitstream for each CTU and may have different values for the luma and the two chroma components of each CTU [7]. In particular, the *SAO Type* parameter signals the decoder which SAO filtering should be applied (none, band offset or edge offset). Nevertheless, the SAO filter can be disabled/enabled at frame-level, where the chosen frames are selected on the encoder side.

When the *SAO Type* parameter is equal to the band offset mode, the full amplitude of the sample range is divided by 32, in order to define a set of *bands*. The filtering procedure for this mode consist of adding an offset value to all samples whose values belong to the same *band*. For example, in Fig. 3a, the deblocked samples from

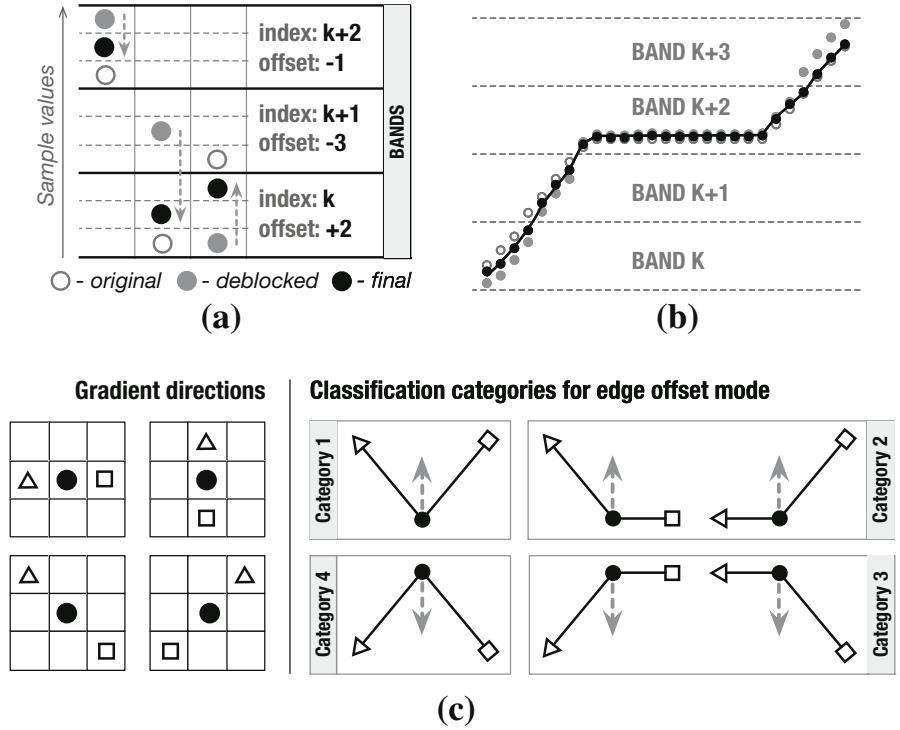

Fig. 3 SAO Band Offset and Edge Offset modes **a** SAO Band Offset filtering, **b** SAO Band Offset example and **c** Gradient directions and categories for the SAO Edge Offset

bands with indexes k, $k+1$ and $k+2$ are added to offset values of $+2$, -3 and -1, respectively, in order to push the final sample values towards the original ones. To reduce the complexity, in the HEVC standard, only four consecutive *bands* are considered for SAO band offset filtering. In this way, only the lowest *band* index needs to be stored in the bitstream, namely the *SAO Band Position* (k in Fig. 3a). For each processed band, a single offset value is provided in the respective *SAO Offset Value* parameter. In Fig. 3b, an example of corrupted deblocked samples by quantization errors are presented in gray-filled dots, where the horizontal and vertical axis denote sample spatial position and value, respectively. In this case, the final filtered samples (dark-filled dots, in Fig. 3b) from *bands* k to $k+3$ are corrected with the SAO band offset filtering by moving towards to the original samples (white-filled dots).

Regarding the edge offset SAO mode, the decoded CTU samples are classified into four categories according to the corresponding gradient direction, as specified in the *SAO Edge Class* parameter. Figure 3c depicts all four possible gradient directions and allowed SAO categories. Similarly to the band offset mode, the offset value assigned to each category is stored in the *SAO Offset Value* parameter. The *SAO Offset Value* is positive for categories 1 and 2 and negative for categories 3 and 4 (see arrows in Fig. 3c). Hence, whenever a sample is classified in one of these categories, its deblocked sample is added to the corresponding *SAO Offset Value*.

3 Proposed Parallel In-Loop Filters

In this section, the proposed parallelization is described. First, the frame-level decoupling of the CPU implementation is presented. Then, the GPU algorithms of the DBF and SAO are elaborated. The proposed approaches are designed to efficiently exploit the computational potential of GPU architectures. They leverage the fine-grain parallelism of each sub-modules and provide fully compliance to HEVC decoding.

3.1 CPU Frame-Decoupled (CFD) In-Loop Filters

The state-of-the-art HEVC decoder proposed in [3] exploits Single Instruction, Multiple Data (SIMD) instructions and data locality to effectively improve the overall performance. The data locality is increased by executing all HEVC modules at CTU level. In this way, all HEVC decoding procedures are performed sequentially inside a CTU, e.g., all possible 8×8 borders in a CTU are filtered as soon as the reconstructed CTU is obtained, where the intermediate data from the previous procedure is directly reused by the next decoding phase. The key advantage of this approach is that this intermediate data for one CTU is rather small, which can be easily accommodated in the CPU cache memory space and the memory bandwidth required to the off-chip memory is reduced.

Such block-based CTU-level implementation, however, is not appropriate for GPU parallelization due to the insufficient parallelism. To exploit the throughput-oriented design of GPUs, the GPU kernels are applied at frame level. However, the difference in data granularity between CPU and GPU makes a direct comparison unfair. Therefore, a new in-loop filters approach for CPU, which applies the processing at frame level,

is proposed herein as CPU Frame-Decoupled (CFD). In practice, the kernels of DBF and SAO are decoupled from the original decoding loop and their kernel inputs are collected into corresponding input buffers. Afterwards, when the frame reconstruction is complete, the DBF is applied for the entire frame with the collected input. Finally, the SAO filter is performed in the deblocked frame to produce the final result. Naturally, such algorithm compromises data locality, because the granularity increases from CTU level to frame level. Nevertheless, frame-level parallelization approaches have already demonstrated the viability of this strategy [4].

3.2 GPU Frame-Decoupled (GFD) In-Loop Filter

The GPU execution is organized in groups of 32 parallel threads (or *Warps*, in NVIDIA's terminology). They are in turn grouped in several Thread Blocks (ThBs). To maximize the attained performance of the in-loop filters for GPU devices, the proposed algorithms carefully maximize the number of active warps, while ensuring that all threads in a warp perform the same operation from the GPU code (kernel). Moreover, the data accesses were carefully designed, in order to efficiently map the HEVC in-loop filters to the GPU memory hierarchy (i.e., global, cache, shared and constant memory). Finally, Compute Unified Device Architecture (CUDA) programming model [14] is used to implement the in-loop filters on GPUs.

3.2.1 Proposed GPU-Based Deblocking Filter

The DBF module consists of two filters, i.e., the horizontal filter and the vertical filter, as shown in Fig. 2. According to the HEVC standard [10], their execution has to be in order. All vertical edges in a frame have to be applied by the horizontal filter first, followed by the vertical filter for all horizontal edges. If these two filters were separately implemented in different kernels, then two kernel launches would be required, leading to kernel launch overheads and redundant data accesses to the intermediate result in global memory.

To circumvent these limitations, both [16] and the proposed implementation consider the fusion of these two stages into one single kernel. Thus, only one kernel launch is needed and redundant accesses to global memory can be avoided. This is possible because these two consecutive filters can be independently applied at 8×8 block samples, as shown in Fig. 4. The independence is guaranteed since both filters need (at most) four input samples and the filtering output affects up to three samples in each side of the edge. Hence, these 8×8 sample blocks, herein referred to as Boundary Blocks (BBs), allow performing both horizontal and vertical filtering on a small subset as long as their execution order (first horizontal filter then vertical filter) is preserved.

As it is shown in Fig. 5, shared memory is used in [16] with the purpose of reducing the required data transfers from and to the GPU global memory between horizontal and vertical filtering. However, since the horizontal and the vertical filters are not always jointly enabled, another approach has been addressed in the proposed work. Figure 6 shows the adopted design without shared memory, which performs the DBF only when needed. When both the horizontal and the vertical filters are disabled, the kernel does

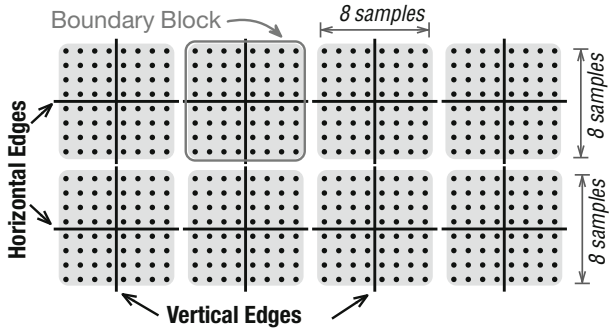

Fig. 4 Edge-level parallelism exploited by the proposed GPU deblocking filtering algorithm and [16]

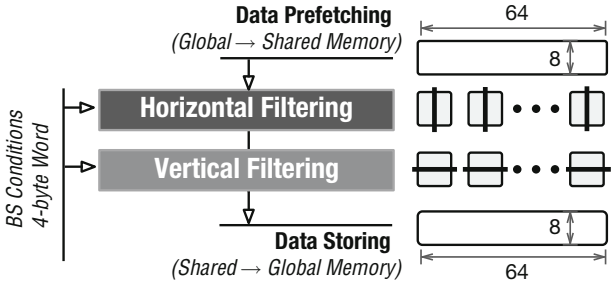

Fig. 5 Warp-level processing for the GPU implementation in [16]

nothing. Otherwise, if either horizontal or vertical filter is enabled, the data is directly loaded from the global memory to the register file, applied with corresponding filter, and stored back to global memory. When both of them are enabled, the intermediate results (after horizontal filter) are stored and loaded again from global memory. In either case, the proposed design achieves a higher performance. Naturally, if both filters are enabled, temporal locality can be exploited with GPU cache, since they are performed consecutively.

Moreover, the proposed approach also adopts a different thread mapping, in order to increase the number of BBs to be processed by a warp. In [16], each warp was mapped to an area of 64×8 samples, in which each thread is mapped to an edge area of 8×4 or 4×8 samples, as shown in Fig. 7. Under this circumstance, however, only 16 edges can be filtered in parallel. On the other hand, the new thread mapping that is now proposed has been improved to process more edges in parallel. Compared to Fig. 7, the size of the thread block behalves with two warps only, but maps to the same size of 256×8 samples, as shown in Fig. 8. These two warps collaboratively perform the deblocking kernel. When processing the horizontal filter, each warp maps to 256×4 samples, where each thread maps to one horizontal edge of 8×4 samples. When processing the vertical filter, each warp maps to 128×8 samples, where each thread maps to one vertical edge of 4×8 samples. Because of the different thread mapping between these two filter stages, a synchronization step is required in between, as a compromise to exploiting the fine grain parallelism for each filtering stage.

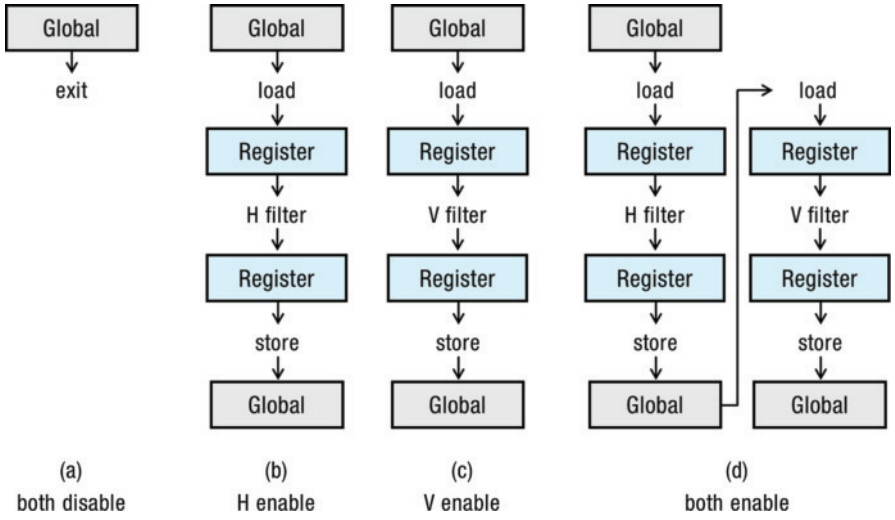

Fig. 6 Data flow without shared memory for the four possible cases

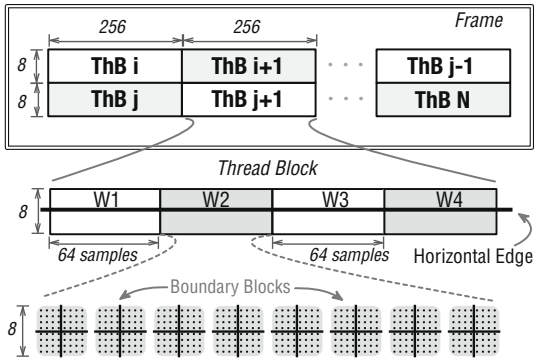

Fig. 7 Thread block assignments for one frame, consisting of four warps per ThB and eight BBs per warp (Wi) [16]

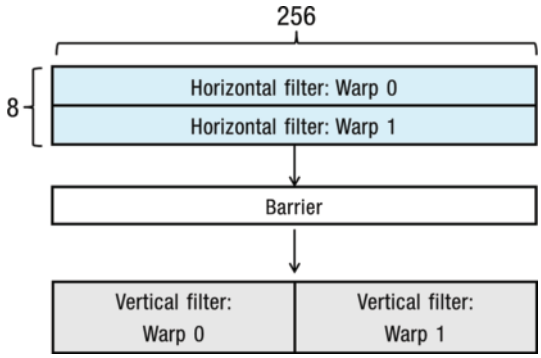

Fig. 8 Thread mapping switch between horizontal and vertical filters with a synchronization barrier in between

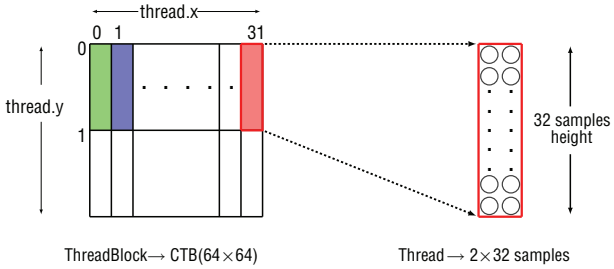

Fig. 9 SAO thread mapping, where each thread block is mapped to one CTB with 64×64 samples and each thread operates on 2×32 samples

Despite the distinct processing scheme, the proposed DBF approach supports the case when the (QP) varies within a picture, as specified in the HEVC standard [10], since the activation of deblocking filter also depends on QP [13]. The QP value is directly obtained from the bitstream and may vary on a basis of the coding units, whose minimal size is 8×8 samples. Therefore, another buffer is allocated, in which each QP value is stored in one byte and corresponds to a block of 8×8 samples. Within this byte, only 6 bits are used, since the QP value ranges from 0 to 56.

3.2.2 Proposed GPU-Based Sample Adaptive Offset

The proposed parallel implementation of the SAO algorithm adopts a thread mapping as shown in Fig. 9, where each thread block is mapped to one CTU. Within each thread block, two warps are configured, which correspond to the top and bottom half of one CTU, respectively. Within each warp, each thread is mapped to an area of 2×32 samples.

The proposed approach reduces the GPU global memory accesses for the luma plane by half, since each thread maps to two samples apart (see Fig. 9). In this way, the entire row of CTUs can be loaded with one single memory access, instead of two. Furthermore, the proposed SAO algorithm implementation exploits GPU vector instructions [14] to increase the parallelism. With a vector length of two, each thread can simultaneously process two samples. Hence, to process its mapped areas, each thread iterates the vector operations for 32 times. This approach also facilitates the thread mapping when processing the chroma planes with 4:2:0 chroma subsampling format. At the horizontal direction, the thread index of luma plane is divided by 2, which can be implemented by a simple right shift. In the vertical direction, the times of iteration (32 in luma) also behalves, which can also be derived with a shift operation.

For CTUs with edge offset mode, another optimization is applied to save computations. To determine the corresponding offset, each sample needs to calculate its offset index. The index, in turn, depends on the difference to its neighboring samples, as shown in Eqs. (1), (2), and (3). Because each thread processes more than one sample, the procedure to compute the index can be shared. The case for index calculation sharing in the horizontal direction is presented in Fig. 10.

Fig. 10 Index calculation sharing between neighboring samples

$$offset(index) = \begin{cases} offset_0, & index = -2 \\ offset_1, & index = -1 \\ 0, & index = 0 \\ offset_2, & index = 1 \\ offset_3, & index = 2 \end{cases} \tag{1}$$

$$index(x, y) = \begin{cases} s(P_{x,y} - P_{x-1,y}) + s(P_{x,y} - P_{x+1,y}), & eo_0 \\ s(P_{x,y} - P_{x,y-1}) + s(P_{x,y} - P_{x,y+1}), & eo_1 \\ s(P_{x,y} - P_{x-1,y-1}) + s(P_{x,y} - P_{x+1,y+1}), & eo_2 \\ s(P_{x,y} - P_{x+1,y-1}) + s(P_{x,y} - P_{x-1,y+1}), & eo_3 \end{cases} \tag{2}$$

$$s(n) = sign(n) = \begin{cases} -1, & n < 0 \\ 0, & n = 0 \\ 1, & n > 0 \end{cases} \tag{3}$$

This way, the two samples in green (*00* and *10*) represent the thread's mapped samples in the same line. Their index calculation is indicated by arrows, where each arrow stands for a sign operation, as shown in Eq. (3). It can be seen that the right sign of *00* can be shared with the left sign of *10*, but with a negated value. Similarly, the index calculation can also be shared in the vertical direction. In fact, this sharing is more relevant in the vertical direction, since there are 32 samples per thread in this direction.

Another optimization involved in the developed SAO filter is concerned with the fact that the kernel is invoked only when needed. Hence, the final output buffer is either written by the deblocking filter or by the SAO filter, depending on the activation of SAO for specific frames.

4 Experimental Evaluation

In this section, the performance of the proposed parallel implementations of the HEVC in-loop filters is experimentally evaluated according to the recommended JCT-VC test conditions [1], namely:

- **HEVC Profile**: Main (8-bit depth with 4:2:0 chroma subsampling);
- **Video Class**: A (2560×1600), B (1920×1080) and E (1280×720);
- **All Intra** and **Random Access** configurations;
- **QPs**: 22, 27, 32, 37.

For such purpose, a set of encoded bitstreams, corresponding to the highest and most common frame resolutions (classes A, B and E) were considered, since they are the most computationally demanding. An additional set of Ultra HD 4K (3840×2160) video sequences [8], referred as *class S*, was also evaluated. Moreover, the maximum nominal sequence frame rate per class is 50 FPS for class S and 60 FPS for classes A, B and E. The selected video sequences were encoded with the HM 15.0 reference

Table 1 Evaluation system setups

Desktop		Embedded	
CPU	GPU	CPU	GPU
Intel i7-6700K	NVIDIA TITAN X	ARM Cortex-A15	NVIDIA GK20a
Haswell	Maxwell	ARMv7-A	Kepler
4.00 GHz	1.08 GHz	2.32 GHz	0.85 GHz
AVX2	–	NEON	–
–	CUDA 7.5	–	CUDA 6.5

software [11] according to [1]. In order to simulate the worst case scenario, neither coding option of Tiles nor Wavefront Parallel Processing (WPP) is considered.

In what concerns the used configurations, the *Random Access* mode was chosen because it is the most common one, for which the frames are organized in a pyramidal structure with *I* and *B* frames. In particular, while *I* frames are encoded with only intra prediction capabilities, the PUs of *B* frames can be intra or inter predicted. Moreover, the *All Intra* configuration, where all frames are *I* frames, is used herein to simulate the worst configuration case. In this case, since the intra prediction may result in an increased amount of residual data, the probability of blocking artifacts and sample distortions also increases, which, on the other hand, increases the computational load of the in-loop filters.

The experimental results were obtained on two different platforms, i.e., a state-of-the-art desktop machine and an embedded development board, as presented in Table 1. The desktop system includes an Intel Haswell CPU and an NVIDIA Maxwell GPU with the latest CUDA version 7.5. The embedded platform is the NVIDIA Jetson TK1 System on Chip (SoC) with a Kepler GPU. In this case, CUDA version 6.5 was used due to limitations of the official firmware. All CPU versions were optimized with SIMD vector instructions, where AVX2 was applied for the desktop and NEON for the embedded system.

Since the CPU and the GPU share the same memory space in the embedded development board, the input data required to perform the GPU HEVC in-loop filters is directly obtained from the SoC main memory through the CUDA zero copy instruction. Due to the limited compute capability of the embedded GPU, the GPU kernel configurations (i.e. the number of warps per ThB, the usage of shared memory and registers, etc.) must be carefully chosen, in order to maximize the number of active warps in the NVIDIA Kepler Streaming Multiprocessor (SM). Furthermore, only one CPU core (no multithreading) was used for the sake of this evaluation and relative assessment.

In order to better showcase the capabilities of the proposed approach, an extensive experimental evaluation was conducted, which tackles different execution approaches. First, a deep profiling analysis of the proposed in-loop filter algorithms *CFD* (CPU) and *GFD* (GPU) is presented for both system environments. Afterwards, the overall performance of the proposed approach is presented (in FPS). For a specific set of video sequences (e.g., with the same QP, video class, etc.), the average frame rate is derived as the total number of frames in a set divided by the aggregated decoding

Table 2 Average processing time (in milliseconds) and obtained speedup per HEVC in-loop filters in the desktop machine

Class	COpt [3]		CFD		GOpt [16]		GFD	
	SAO	DBF	SAO	DBF	SAO	DBF	SAO	DBF
All Intra configuration								
S (3840 × 2160)	1.48	4.87	2.75	6.71	0.38	0.41	0.23	0.29
A (2560 × 1600)	0.72	2.42	1.22	2.62	0.21	0.23	0.15	0.16
B (1920 × 1080)	0.39	1.47	0.61	1.55	0.19	0.14	0.09	0.10
E (1280 × 720)	0.18	0.61	0.26	0.66	0.15	0.07	0.07	0.05
Average speedup	1×	1×	0.6×	0.8×	3.0×	11.0×	5.1×	15.6×
Random Access configuration								
S (3840 × 2160)	1.27	2.80	1.74	3.39	0.30	0.51	0.14	0.16
A (2560 × 1600)	0.57	1.33	0.79	1.40	0.16	0.28	0.08	0.09
B (1920 × 1080)	0.26	0.60	0.33	0.62	0.13	0.15	0.04	0.05
E (1280 × 720)	0.10	0.16	0.07	0.14	0.10	0.07	0.01	0.02
Average speedup	1×	1×	0.8×	0.9×	3.2×	4.8×	8.1×	15.3×

time. Furthermore, the proposed design is compared with the state-of-the-art GPU-based [16] (*GOpt*) and CPU-based [3] (*COpt*) HEVC in-loop filters implementations.

4.1 Profiling

Tables 2 and 3 present the performance result of the four versions of HEVC in-loop filter for the desktop and the embedded setups, respectively. For video sequences within a single class, the performance is reported as the average processing time per frame considering all QPs (i.e., from 22 to 37). Moreover, the results are separated by the two used configurations, i.e., *All Intra* and *Random Access*. As it was expected, the average processing time per frame obtained with all considered CPU and GPU HEVC parallel modules varies across different classes. Naturally, the highest per-module time was observed for the highest resolution frames due to the increased amount of data to be processed. Furthermore, for all tested video sequences and QP values, the DBF usually represents the most time consuming in-loop filter, due to its higher computational load. In contrast, the SAO module exploits a higher amount of data parallelism, thus leading to a lower processing time, when compared to the DBF.

When compared with COpt [3], the proposed CFD version does not attain the same performance due to the lost of locality in all configurations, except for sequences of *class E* with *Random Access* configuration in both execution environments. For this particular case, as presented in Table 2, the COpt achieves 0.10 and 0.16 ms, while the CFD can filter a frame at 0.07 and 0.14 ms, for the SAO and DBF, respectively. In this case, the penalties resulting from the loss of locality are reduced, since *class E* videos have the smallest frame resolution among all tested video sequences. Moreover, the SAO filtering is not performed in most of the frames with *Random Access* configura-

Table 3 Average processing time (in milliseconds) and obtained speedup per HEVC in-loop filter in the NVIDIA Jetson TK1 development board

Class	COpt [3]		CFD		GOpt [16]		GFD	
	SAO	DBF	SAO	DBF	SAO	DBF	SAO	DBF
All Intra configuration								
S (3840 × 2160)	12.75	25.81	24.53	44.23	5.18	10.29	6.20	6.94
A (2560 × 1600)	7.63	13.20	13.16	17.76	2.87	5.52	3.22	3.59
B (1920 × 1080)	4.26	7.72	6.78	9.01	1.59	3.07	1.72	2.10
E (1280 × 720)	1.91	3.25	2.43	3.58	0.69	1.46	0.83	0.96
Average speedup	1×	1×	0.6×	0.7×	2.6×	2.5×	2.2×	3.7×
Random Access configuration								
S (3840 × 2160)	10.51	14.76	15.28	20.98	4.40	13.09	3.81	4.14
A (2560 × 1600)	5.20	6.96	7.31	8.28	2.20	6.77	1.79	2.21
B (1920 × 1080)	2.30	2.99	2.92	3.14	1.15	3.30	0.79	1.00
E (1280 × 720)	0.77	0.73	0.57	0.66	0.45	1.39	0.20	0.25
Average speedup	1×	1×	0.7×	0.8×	2.3×	1.0×	2.8×	3.3×

tion. This fact provides the performance improvements of the CFD implementation, since those frames can be skipped due to the frame-level processing approach. On the other side, the average filtering time in COpt has taken into account the required memory transfers from a CTU-based buffer to the final decoded picture buffer, because of its CTU-level processing design and the fact that the SAO filter is the last stage in the decoding procedures. Regarding the DBF module with CTU-based filtering of COpt, memory copies also have to be considered, in order to maintain both filtered and unfiltered data at the boundary, since the intra prediction procedure uses unfiltered samples. In contrast, due to the frame-level processing in both CFD and GFD algorithms, the intra prediction procedure for the entire frame has to be executed before the in-loop filtering, where filtered and unfiltered data maintenance for the boundaries is not required anymore.

Among all tested in-loop filter approaches, the proposed GFD achieves the best performance for all classes and configurations in the desktop environment (see Table 2). In particular, when compared to COpt, the average speedup of 8.1× and 15.6× was achieved for the SAO and DBF, respectively. GFD also achieves a higher performance than the GOpt implementation, mainly due to the reduced memory transactions for DBF procedures, and the usage of vector instructions in the SAO filtering, which increase the number of samples processed in parallel. However, in the NVIDIA Jetson TK1 (see Table 3), the performance of the GFD SAO filter is penalized due to its reduced occupancy on Kepler architecture. For both Kepler and Maxwell architectures, the maximum allowed number of warps per SM is 64, while their maximum allowed number of thread blocks differs, with 16 on Kepler versus 32 on Maxwell. In other words, the Maxwell architecture is more occupancy friendly to kernels with smaller thread block size, where only 2 warps per thread block are sufficient to reach full occupancy, instead of 4 warps on Kepler. Therefore, the occupancy of SAO kernel in

GFD (with a block size of 2 warps) behalves on Kepler while in GOpt it remains the same, where a thread block size with 4 warps is configured. The smaller block size of SAO in GFD can be considered as an indirect result because of the vector instruction optimization (with a vector of 2), thus doubling the thread block size would probably help for Kepler but result in a lower performance for Maxwell. Since Kepler is an older architecture and Maxwell is used in the Jetson TX1, the successor of Jetson TK1, further effort has not been put to address this performance portability issue. Because of its reduction in occupancy, GFD achieves an average speedup of 2.2× for *All Intra* configuration, while a speedup of 2.6× is observed for GOpt when compared with COpt. Nevertheless, the proposed GFD algorithm achieves higher performance than GOpt for the *Random Access* configuration, where the SAO filter is not used for the majority of the frames. In this case, the proposed GFD algorithm bypassed the deblocked frame as the final output by updating a memory pointer and without transferring the data.

4.2 Overall Performance

In order to further evaluate the performance of the proposed HEVC in-loop filters, Figs. 11 (desktop machine) and 12 (development board) present the experimentally obtained average frame rate for each considered configuration and resolution. Herein, the average frame rates (in FPS) are obtained for different QP values and for all considered in-loop filter implementations, namely, CFD, GFD, COpt [3] and GOpt [16].

As expected, in both considered environments (desktop and embedded setups) and configurations (*All Intra* and *Random Access*), the performance decreases with the increase of the video resolution. Nevertheless, in all setups and configurations, both GPU versions are able to achieve frame rates above the nominal values by considering only the HEVC in-loop filters (see Figs. 11 and 12). Moreover, the GFD approach outperforms all other HEVC in-loop filters implementations, for all configurations and environments.

When comparing the obtained performance of *All Intra* and *Random Access* configurations, better results are achieved for the latter one for all in-loop filter implementations, except the GOpt. In this case, since the SAO filtering is not used in most of the frames, the GOpt approach is limited by the GPU memory transfers, in order to copy the frame. In this sense, a better overview of the real contributions of the proposed work can be seen in the *All Intra* configuration, where all frames have at least one CTU filtered by the SAO procedure.

For *All Intra* configuration within a single class (e.g., Fig. 11a or 12a), the obtained frame rate corresponding to the herein proposed GFD algorithm only slightly varies with the QP, which contrasts with the *Random Access* configuration. Such interesting phenomena happens in *All Intra* configuration because the SAO filtering is more computationally demanding for lower QP values (or high bitrates), while the DBF has higher computational load for higher QP values (or low bitrates). When operating in lower bitrates, a typical HEVC encoder tends to favor bitrate over distortion, in order to achieve higher bitrate savings. This leads to an increased presence of blocking artifacts, which are more visible for larger QPs, thus increasing the computation demand for

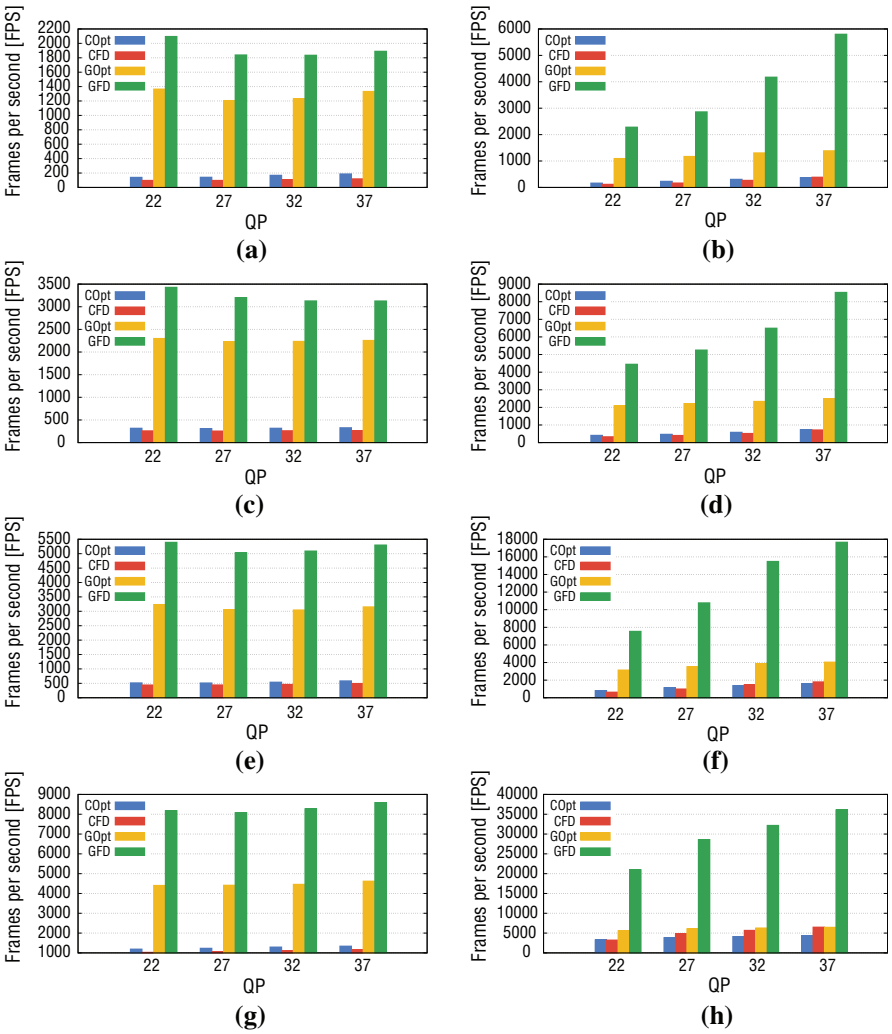

Fig. 11 Average frame rate obtained with the tested GPU HEVC in-loop decoding modules (DBF + SAO) on the desktop machine. **a** Class S—*All Intra* configuration. **b** Class S—*Random Access* configuration. **c** Class A—*All Intra* configuration. **d** Class A—*Random Access* configuration. **e** Class B—*All Intra* configuration. **f** Class B—*Random Access* configuration. **g** Class E—*All Intra* configuration. **h** Class E—*Random Access* configuration

the HEVC DBF module on the decoder side. On the other hand, the SAO module is more computational demanding for lower QPs, due to the increased details for higher spatial sample frequencies, i.e., more visual details obtained with higher bitrates.

In *Random Access* configuration, the GFD exhibits a significant performance gain over the other approaches for all considered QPs, classes and environments. For a fixed class and environment, the best performance is obtained for the highest QP mainly

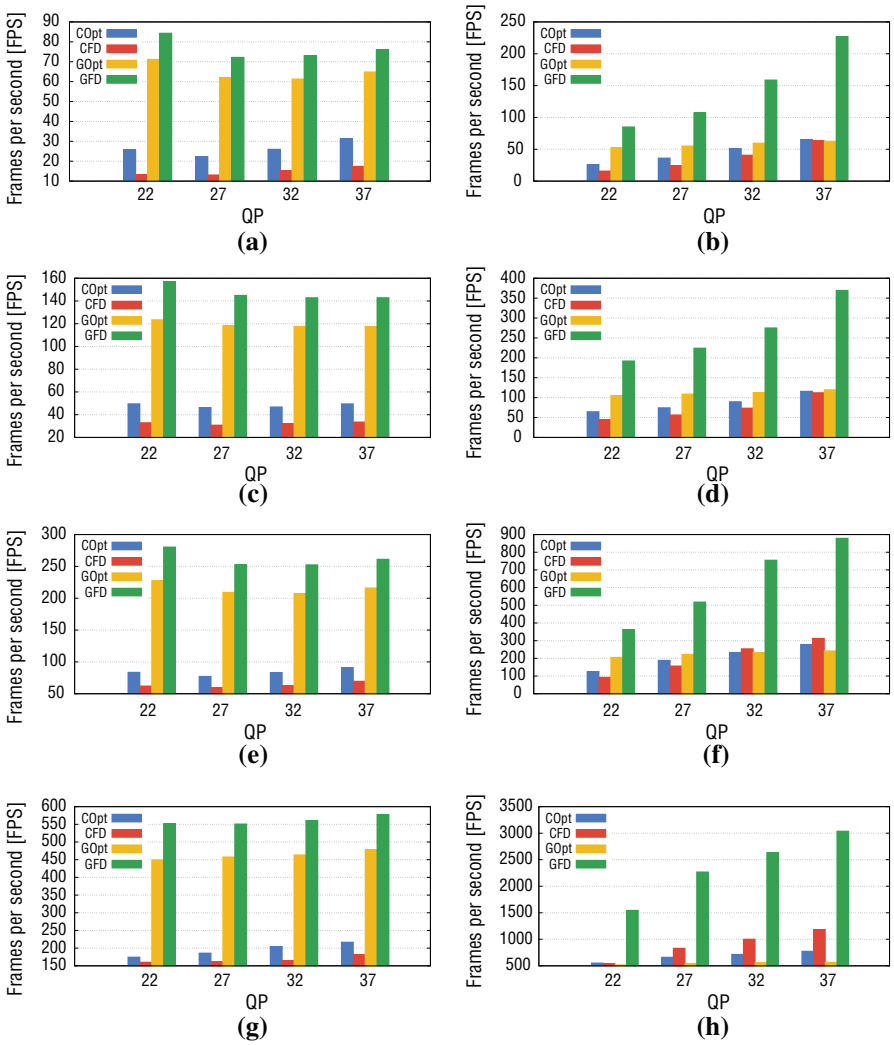

Fig. 12 Average frame rate obtained with the tested GPU HEVC in-loop decoding modules (DBF + SAO) on the NVIDIA Jetson TK1. **a** Class S—*All Intra* configuration. **b** Class S—*Random Access* configuration. **c** Class A—*All Intra* configuration. **d** Class A—*Random Access* configuration. **e** Class B—*All Intra* configuration. **f** Class B—*Random Access* configuration. **g** Class E—*All Intra* configuration. **h** Class E—*Random Access* configuration

because of two reasons. First, SAO is disabled for a majority of frames with higher QP values, where the GFD, CFD and COpt can take advantage of this by memory pointer manipulations. Second, the computational load of DBF is reduced for higher QP values at *Random Access*, in contrast to *All Intra* configuration. In this case, the encoder prioritizes larger PU and TU sizes more frequently to provide bitrate reduction. This fact results in less borders to filter since the DBF is applied in 8×8 grid which relies in PU and TU borders. The GFD approach has been designed to take advantage

of those particularities by avoiding unnecessary computations and memory transfers. In contrast, GOpt does not explicitly consider these two particularities, thus yielding a near-constant performance across different QP values due to its memory-bound nature. Hence, although both the COpt and the CFD can take advantage of these two *Random Access* peculiarities, their performance cannot surpass the one obtained in GFD due to a lower degree of parallelism. Furthermore, the COpt and CFD do not outperform the GOpt implementation except in particular cases when the parallelism degree is not too high (e.g., class E in Fig. 12h) or if the memory bandwidth provided by the GPU is too low (e.g., Fig. 12b, f).

When comparing the achieved performance of the proposed GFD algorithm and COpt in different classes (e.g., Fig. 11a, c, e, g), higher speedups are obtained for higher frame-resolution sequences, due to the increased amount of computational load and parallelism exploited. Finally, it can also be observed that the proposed GFD implementation can handle frame processing time for all video sequences, which could allow real-time performance, i.e., a frame rate of at least 60 FPS is always achieved in both environment setups, configurations and all tested bitrates. In particular, the proposed GFD algorithm allows achieving, in the NVIDIA Jetson TK1 and *All Intra* configuration, an average frame-rate of 76.1 FPS for class S, 146.8 FPS for class A, 261.8 FPS for class B and 558.7 FPS for class E. Thus, it demonstrates the feasibility of effectively accelerating the in-loop decoding modules by using either state-of-the-art GPUs or embedded GPU devices. In this scenario, an optimized CPU implementation of the HEVC decoder could handle the other video decoder modules, while offloading the execution of HEVC in-loop filters to the GPU.

5 Related Work

Along the past years, several video encoding and decoding procedures have been accelerated in not only high performance CPU and GPU devices, but also in low power devices [e.g., Field-Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs)]. In particular, the herein proposed CPU approach is based on a state-of-the-art HEVC decoder from [3], which exploits SIMD parallelism (e.g., AVX2 and NEON) to implement HEVC decoder modules by specifically focusing on modern multi-core CPU architectures, including ARM processors. At the end, an average frame rate of 543, 35.5 and 77.8 FPS for Full HD video sequences was reported with a Haswell, an ARM Cortex-A9 and an ARM Cortex-A15, respectively.

In what concerns GPU devices, the authors in [19] have presented an optimized GPU implementation of the inverse transform and of the motion compensation procedures in [20] for the H.264/MPEG-4 AVC standard. Regarding the HEVC in-loop filters (i.e., DBF and SAO), frame-level optimizations for embedded GPUs have been proposed in [16], where an Ultra HD 4K intra frame is filtered in less than 20 ms for the NVIDIA Jetson TK1 development board. In particular, the GPU algorithms presented herein further improve the implementation from [16] by optimizing the memory accesses and by including vector instructions, specially for desktop GPUs.

When considering individual filters, a GPU-based DBF has been proposed in [6], where an average performance of 200 FPS and 333 FPS was achieved for *All Intra* and *Low Delay* configurations, respectively, in the NVIDIA GeForce 710M GPU. In [17], the authors decreased the frame-level parallelism for the SAO procedure by including it in the CTU decoding procedure, in order to better exploit memory-bandwidth and cache performance. A similar design is proposed in [3] (for CPUs) and in [9], which presents a very low-power programmable coprocessor architecture targeting especially embedded devices.

When looking at different approaches for portable devices, specific hardware for HEVC in-loop filters has been proposed in [5] and [21]. However, such implementations usually represent different compromises in terms of programmability, resources utilization and energy efficiency, thus preventing a fair comparison with high-performance computing devices, like GPUs.

6 Conclusions

In this paper, an efficient implementation of the HEVC in-loop filtering modules (DBF and SAO) has been proposed to reduce their decoding time on GPU devices (referred to as GFD). When compared to previous work, the proposed implementation and optimizations result in higher performance due to the increased amount of parallelism and reduced memory transfers. In addition, a CPU-based frame-level in-loop filter (referred to as CFD) was also developed, in order to provide a more fair comparison across CPU and GPU architectures.

When compared with previous GPU-based approaches, the implemented DBF has been redesigned without shared memory, to avoid unnecessary data transfers when the borders are not filtered. Moreover, the GPU thread assignment of the DBF kernel in the GFD implementation has been improved to enable more parallelism, to efficiently exploit the GPU resources, and to increase the number of active warps. In the SAO filter, both CFD and GFD approaches exploit the frame-level processing and the fact that not all frames in the sequence are filtered. Furthermore, the SAO in the GFD implementation has been designed to exploit the intrinsic GPU vector instructions, thus further boosting the performance of this module.

The proposed approach has been experimentally evaluated on a state-of-the-art desktop and on an embedded system. The obtained results show that the GFD approach outperforms the current state-of-the-art CPU and GPU HEVC in-loop filters for all tested configurations, recommended bitrates and platforms. For example, on the NVIDIA GTX TITAN X, it achieves a speedup of 1.6× for *All Intra* configuration and 2.9× for *Random Access* configuration, when compared to GOpt. On the NVIDIA Jetson TK1 development board with limited computational resources, the proposed GFD approach delivers an average processing rate higher than the nominal frame rate of Ultra HD 4K sequences (50 FPS), which is also the most computationally demanding video class. In particular, the proposed approach provides an average frame rate of 76 FPS for *All Intra* configuration and 125 FPS for *Random Access* configuration.

Acknowledgements This work was supported by national funds through FCT, under projects PTDC/EEI-ELC/3152/2012 and UID/CEC/50021/2013. Diego F. de Souza also acknowledges FCT for the Ph.D. scholarship SFRH/BD/76285/2011.

References

1. Bossen, F.: Common test conditions and software reference configurations. Doc. JCTVC-L1100 of JCT-VC (2013)
2. Bossen, F., Bross, B., Suhring, K., Flynn, D.: HEVC complexity and implementation analysis. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1685–1696 (2012). doi:10.1109/TCSVT.2012.2221255
3. Chi, C.C., Alvarez-Mesa, M., Bross, B., Juurlink, B., Schierl, T.: SIMD acceleration for HEVC decoding. IEEE Trans. Circuits Syst. Video Technol. **25**(5), 841–855 (2015). doi:10.1109/TCSVT.2014.2364413
4. Chi, C.C., Alvarez-Mesa, M., Juurlink, B., Clare, G., Henry, F., Pateux, S., Schierl, T.: Parallel scalability and efficiency of HEVC parallelization approaches. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1827–1838 (2012). doi:10.1109/TCSVT.2012.2223056
5. Cho, S., Kim, H., Kim, H.Y., Kim, M.: Efficient in-loop filtering across tile boundaries for multi-core HEVC hardware decoders with 4 K/8 K-UHD video applications. IEEE Trans. Multimed. **17**(6), 778–791 (2015). doi:10.1109/TMM.2015.2418995
6. Eldeken, A.F., Dansereau, R.M., Fouad, M.M., Salama, G.I.: High throughput parallel scheme for HEVC deblocking filter. In: 2015 IEEE International Conference on Image Processing (ICIP), pp. 1538–1542 (2015). doi:10.1109/ICIP.2015.7351058
7. Fu, C.M., Alshina, E., Alshin, A., Huang, Y.W., Chen, C.Y., Tsai, C.Y., Hsu, C.W., Lei, S.M., Park, J.H., Han, W.J.: Sample adaptive offset in the HEVC standard. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1755–1764 (2012). doi:10.1109/TCSVT.2012.2221529
8. Haglund, L.: The SVT high definition multi format test set. Tech. rep., Sveriges Television AB (SVT), Sweden (2006). ftp://vqeg.its.bldrdoc.gov/HDTV/SVT_MultiFormat/SVT_MultiFormat_v10.pdf
9. Hautala, I., Boutellier, J., Hannuksela, J., Silvén, O.: Programmable low-power multicore coprocessor architecture for HEVC/H.265 in-loop filtering. IEEE Trans. Circuits Syst. Video Technol. **25**(7), 1217–1230 (2015). doi:10.1109/TCSVT.2014.2369744
10. JCT-VC: High Efficient Video Coding (HEVC). ITU-T Recommendation H.265 and ISO/IEC 23008-2, ITU-T and ISO/IEC JTC 1 (2013)
11. JCT-VC: Subversion repository for the HEVC test model version HM 15.0 (2014). https://hevc.hhi.fraunhofer.de/svn/svn_HEVCSoftware/tags/HM-15.0/
12. Norkin, A., Bjøntegaard, G., Fuldseth, A., Narroschke, M., Ikeda, M., Andersson, K., Zhou, M., Van der Auwera, G.: HEVC deblocking filter. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1746–1754 (2012). doi:10.1109/TCSVT.2012.2223053
13. Norkin, A., Bjontegaard, G., Fuldseth, A., Narroschke, M., Ikeda, M., Andersson, K., Zhou, M., Van der Auwera, G.: HEVC deblocking filter. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1746–1754 (2012)
14. NVIDIA Corporation: NVIDIA®CUDA™ Compute Unified Device Architecture Programming Guide (version 1.0: Jun. 2007 (and subsequent editions))
15. Ohm, J., Sullivan, G., Schwarz, H., Tan, T.K., Wiegand, T.: Comparison of the coding efficiency of video coding standards-including high efficiency video coding (HEVC). IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1669–1684 (2012)
16. de Souza, D.F., Ilic, A., Roma, N., Sousa, L.: HEVC in-loop filters GPU parallelization in embedded systems. In: 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), pp. 123–130 (2015). doi:10.1109/SAMOS.2015.7363667
17. Subramanya, P.N., Adireddy, R., Anand, D.: SAO in CTU decoding loop for HEVC video decoder. In: 2013 International Conference on Signal Processing and Communication (ICSC), pp. 507–511 (2013). doi:10.1109/ICSPCom.2013.6719845
18. Sullivan, G.J., Ohm, J., Han, W.J., Wiegand, T.: Overview of the high efficiency video coding (HEVC) standard. IEEE Trans. Circuits Syst. Video Technol. **22**(12), 1649–1668 (2012). doi:10.1109/TCSVT.2012.2221191
19. Wang, B., Alvarez-Mesa, M., Chi, C.C., Juurlink, B.: An optimized parallel IDCT on graphics processing units. In: Proceedings of the 18th International Conference on Parallel Processing Workshops,

Euro-Par'12, pp. 155–164. Springer, Berlin, Heidelberg (2013). doi:10.1007/978-3-642-36949-0_18. http://dx.doi.org/10.1007/978-3-642-36949-0_18
20. Wang, B., Alvarez-Mesa, M., Chi, C.C., Juurlink, B.: Parallel H.264/AVC motion compensation for gpus using opencl. IEEE Trans. Circuits Syst. Video Technol. **25**(3), 525–531 (2015). doi:10.1109/TCSVT.2014.2344512
21. Zhou, W., Zhang, J., Zhou, X., Liu, Z., Liu, X.: A high-throughput and multi-parallel VLSI architecture for HEVC deblocking filter. IEEE Trans. Multimed. **PP**(99), 1–1 (2016). doi:10.1109/TMM.2016.2537217