Journal of Real-Time Image Processing (2023) 20:43
https://doi.org/10.1007/s11554-023-01300-z

ORIGINAL RESEARCH PAPER

GPU-based parallelisation of a versatile video coding adaptive loop filter in resource-constrained heterogeneous embedded platform

Anup Saha[1] · Nuno Roma[2] · Miguel Chavarrías[1] · Tiago Dias[3] · Fernando Pescador[1] · Víctor Aranda[1]

Received: 15 December 2022 / Accepted: 25 March 2023 / Published online: 13 April 2023
© The Author(s) 2023

Abstract
This paper presents a GPU-based parallelisation of an optimised versatile video decoder (VVC) adaptive loop filter (ALF) filter on a resource-constrained heterogeneous platform. The GPU has been comprehensively utilised to maximise the degree of parallelism, making the programme capable of exploiting the GPU capabilities. The proposed approach enables to accelerate the ALF computation by an average of two times when compared to an already fully optimised version of the software decoder implementation over an embedded platform. Finally, this work presents an analysis of energy consumption, showing that the proposed methodology has a negligible impact on this key parameter.

Keywords GPU · VVC · Video decoder · Embedded · ALF · Filter

1 Introduction

The Joint Video Experts Team (JVET), from ISO/IEC JTC1 and VCEG (Q6/16), released the current state-of-the-art versatile video coding (VVC) standard in July 2020 [1]. Compared to the High-Efficiency Video Coding (HEVC) standard [2], VVC provides the same visual quality with 50% bit rate savings. However, this reduction in bit rate comes at the cost of a significant increase in computational complexity: ×10 more in the encoder and ×2 more in the decoder, with respect to HEVC [3]. This increase in complexity is a crucial limiting factor for both consumer computers and resource-constrained embedded systems.

The greater coding efficiency that VVC offers is achieved thanks to the introduction of several new functionalities compared to HEVC. For instance, the inclusion of an Adaptive Loop Filter (ALF) in the filtering loop is responsible for reducing the coding artefacts and minimising the mean square error between the original and reconstructed samples. However, these in-loop filters significantly increase the computational complexity requirements of a decoder, on average 30% and 40% when decoding on a high-performance general purpose processor (HGPP) and an embedded general purpose processor (EGPP), respectively [4, 5]. According to [5], ALF alone represents an average computational complexity of 5–12% and 12–24% of the total decoding time share on an HGPP and an EGPP, respectively. As a result, it has become a rather challenging research goal to reduce the in-loop filtering time, and more particularly to reduce the ALF filtering time through parallelisation to achieve real-time decoding.

In this scenario, portable embedded systems on chip (SoCs) are being equipped with dedicated hardware to handle encoding and decoding operations. For example, the NVIDIA Jetson AGX Xavier development kit integrates a full-featured HEVC encoder and decoder [6]. However, the operational condition of this type of hardware is highly

✉ Miguel Chavarrías
miguel.chavarrias@upm.es

Anup Saha
anup.saha@upm.es

Nuno Roma
nuno.roma@inesc-id.pt

Tiago Dias
tiago.dias@isel.pt

Fernando Pescador
fernando.pescador@upm.es

Víctor Aranda
victor.arandal@upm.es

1 Universidad Politécnica de Madrid, Madrid, Spain

2 INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal

3 INESC-ID, ISEL, Instituto Politécnico de Lisboa, Lisbon, Portugal

constrained to predefined parameterisations, including video resolution or frame rate, and only to a particular video standard. To address these limitations, a hybrid approach [7] using a central processing unit (CPU) and a graphics processing unit (GPU) can be employed for efficient encoding and decoding on heterogeneous platforms. In accordance, the main contribution of the presented research is a GPU-based parallelisation of an optimised versatile video decoder (VVdeC) ALF filter. The ALF filtering process has been chosen to be processed by the GPU due to being a high computationally demanding process of VVC and involving a high degree of parallelism. The remaining decoding blocks are executed on the CPU. The main contributions of this article are summarised as follows:

- A detailed methodology is presented to exploit GPU for VVdeC ALF. This information is essential not only for different VVdeC blocks but also for other video encoding and decoding applications.
- The VVdeC ALF source code has been redesigned to maximise the degree of parallelism, making the programme capable of exploiting GPU capabilities.
- The presented results show real-time decoding of FHD sequences using CPU+GPU-based hybrid approach. Furthermore, the information from these results is related with an analysis of the energy consumption of the embedded platforms employed.
- The proposed method does not affect or introduce any loss in video quality, and the decoded video output is exactly the same as the reference.

The rest of this manuscript is structured as follows. Section 2 discusses related work on VVC parallelisation. In Sect. 3, a brief overview of a simplified block diagram of the VVC decoder and the ALF filtering process is presented. Section 4 discusses the current implementation of ALF in VVdeC. In Sect. 5, the proposed methodology is detailed. Section 6 presents the experimental results obtained. Finally, the conclusion of the paper is provided in Sect. 7.

2 Related work

To tackle the complexity issues of the VVC standard, several parallel CPU and other implementations based on heterogeneous architectures were proposed for VVC codecs. An optimised VVC decoder was presented in [8] that supports real-time decoding using single instruction multiple data (SIMD) intrinsics and multi-core processing with an x86 architecture. S. Gudumasu et al. [9] proposed a redesign technique of the VVC decoder based on data and task parallelisation that achieved real-time decoding using multi-core CPU over heterogeneous platforms. In [10], the authors used not only data and task parallelisation but also SIMD instructions to accelerate the VVC decoder on x86 CPUs. In [11], an optimised VVC software decoder is proposed for ARM-based mobile devices that exploit the intrinsic and multi-CPU cores of SIMD based on ARM Neon.

Several GPU-based implementations have also been reported in the literature for different video encoders and decoders, including VVC. In [12], the proposed VVC implementation accelerates the VVC reference software decoder by 16× using CPU+GPU. Here, VVC motion compensation was accelerated by organising the GPU threads and by satisfying data dependencies for re-partitioning the coding unit. In [13], a GPU-based implementation of the adaptive multiple transform block is presented. This functionality is also included in VVC, but on top of the Open-HEVC decoder [14]. The study obtained a performance improvement of 11× in the transforms. In [15], Wang et al. introduced a pipeline structure for parallel in-loop filtering in the HEVC encoder using a GPU, achieving up to 47% of time savings in filter performance. In [16] and [17], the authors accelerated HEVC inverse quantisation, inverse transformation, intra-prediction, and in-loop filters using GPUs, with the aim of satisfying real-time requirements by obtaining more than 40 fps for 4K Ultra High Definition (UHD) video sequences. Furthermore, a set of parallel algorithms based on CPU + GPU are presented in [18], where context adaptive binary arithmetic coding (CABAC) is processed on the CPU and inverse quantisation, inverse DCT, intra- and inter-decoder, and in-loop filters are processed on the GPU. This solution achieved real-time decoding for High Definition (HD) sequences with a frame rate of up to 67 fps. Zhang et al. designed several core-based parallel algorithms for the Sample Adaptive Offset (SAO) filter [19]. This implementation achieved a 22× speedup of the SAO process. Regarding the second generation of the Audio Video Coding Standard (AVS2) [20], the loop filter, which includes the deblocking filter (DBF), SAO, and ALF, was also accelerated by 22× using a load balanced implementation of CPU+GPU together with several other memory optimisations [7]. The authors in [21] and [22] presented a GPU-based implementation of intra-decompression and intra-decoding, respectively, for the third generation of the Audio Video Coding Standard (AVS3) [23]. Both implementations obtained real-time decoding for 8K video with an average frame rate of more than 47 fps.

Taking advantage of the similarity in the implementation tools, these techniques can also be applied to accelerate VVC encoder and decoder operations. Up until now, this approach was not followed and most research work focused on optimisations for nonresource-constrained

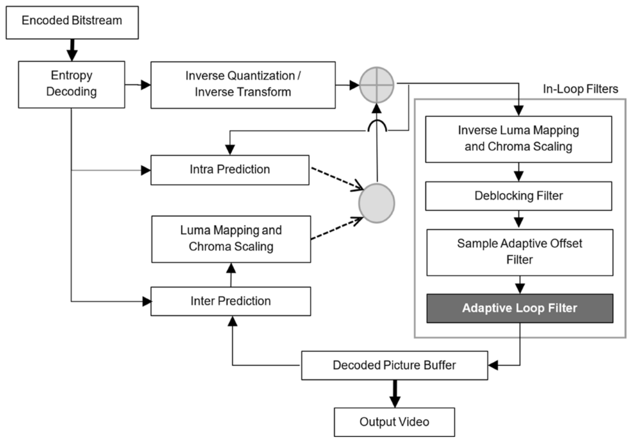

Fig. 1 Simplified block diagram of a VVC decoder

platforms. This work aims to fill this gap using the GPU of a resource-constrained heterogeneous platform to accelerate the computation of the ALF block of an already optimised VVC decoder.

3 VVC decompression

This section introduces the decoder block diagram with a special focus on the optimised ALF module.

3.1 Decoder structure

The simplified block diagram of a standard VVC decoder is presented in Fig. 1. The decoding process begins with the entropy decoding (ED) of the input bitstream through CABAC [24]. The entropy decoding process produces all the information required for decompressing the video. Then, the inverse quantisation and inverse transformation (TX) process produce the residual data from the input coefficients. These residuals are then added to the prediction pixels from intra-prediction (IP) or inter-prediction (EP). Then, four in-loop filters are applied: (1) inverse luma mapping with chroma scaling [25] helps to improve the coding efficiency by efficiently mapping the range of variation of the input signals, (2) DBF is applied at the block boundaries to mitigate the block artefacts, (3) SAO is applied after DBF to reduce the sample distortion, and (4) ALF is used to minimise the mean square error. Finally, after finishing the ALF filtering, the decoded video is obtained.

As presented in the introduction, this work is focused on the parallelisation of the ALF block in GPU, by aiming at the migration of its execution to a GPU kernel. For this reason, some details of this filter are presented below.

3.2 Adaptive loop filtering

ALF is one of the in-loop filters that is applied at the end of the decoding process. It is applied to samples previously filtered by the DBF and SAO filters. The core operation of ALF is based on Wiener filters [26]. It was designed to reduce the mean square error between the original and reconstructed samples, and to reduce the coding artefacts caused by the previous stages. The functionality of the ALF decoder side is illustrated in Fig. 2. It consists of three main processes: (1) classification of luma components based on gradient calculation, (2) filtering of the luma component, and (3) filtering of the chroma component. In addition, cross-component filtering is adapted by VVC ALF.

3.2.1 Classification of luma components

The subblock-level classification process starts by classifying each 4×4 size block into one of 25 classes. This classification is based on directionality and a quantified value that represents the activity of each sample within the block. The

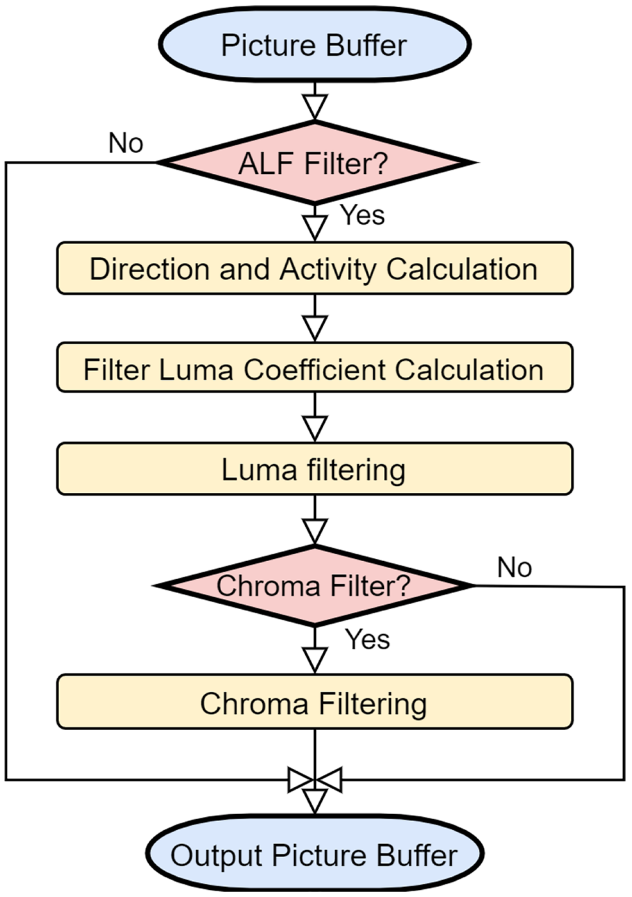

Fig. 2 Diagram of the general working flow of ALF

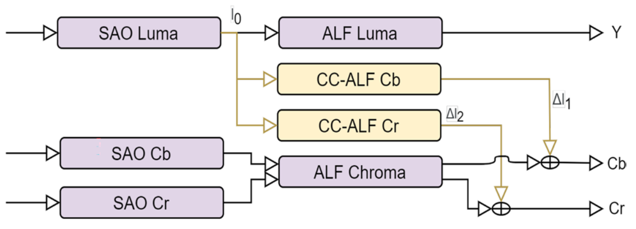

Fig. 3 Diagram of the CCALF architecture

determination of these parameters involves the calculation of the horizontal, vertical, and two diagonal gradients for the reconstructed samples. The calculations involved [27] are shown in Eqs. (1) and (2) for the vertical and horizontal directions and in Eqs. (3) and (4) for the two diagonal directions, where Y is the reconstructed sample and G_v, G_h, G_{d_o}, and G_{d_1} are vertical, horizontal, and two directional gradients, respectively

$$G_v(i,j) = |2Y(i,j) - Y(i-1,j) - Y(i+1,j)| \tag{1}$$

$$G_h(i,j) = |2Y(i,j) - Y(i,j-1) - Y(i,j+1)| \tag{2}$$

$$G_{d_o}(i,j) = |2Y(i,j) - Y(i-1,j-1) - Y(i+1,j+1)| \tag{3}$$

$$G_{d_1}(i,j) = |2Y(i,j) - Y(i-1,j+1) - Y(i+1,j-1)|. \tag{4}$$

3.2.2 Luma and chroma component filtering

After classification, the ALF filter applies the respective coefficients to the reconstructed samples obtained in the output of SAO. VVC ALF considers a 7×7 diamond-shaped (DMS) filter for the luma component (see Fig. 5-left) and a 5×5 DMS filter for the chroma component. Here, each luma or chroma component is represented by a square, while c_i represents a coefficient value. The centre of the square represents the current component to be filtered. Equation 5 [28] is used to calculate the filtered component value $\tilde{Y}(x, y)$ at the (x, y) coordinate

$$\tilde{Y}(x,y) = Y(x,y) + \left(\sum_{i=0}^{N-2} c_i(Y(x+x_i, y+y_i) - Y(x,y)) + \sum_{i=0}^{N-2} c_i(Y(x-x_i, y-y_i) - Y(x,y)) + 64 \right) \gg 7. \tag{5}$$

Here, $Y(x+x_i,y+y_i)$ and $Y(x-x_i,y-y_i)$ represent the component value corresponding to c_i and N represents the number of coefficients. The value of N is 13 for the 7×7 DMS filter and 7 for the 5×5 DMS filter.

3.2.3 Cross-component ALF filtering

The final cross-component ALF filtering (CCALF) refines the chroma component using the values of the luma component. It receives this name, because the input component for the filter operation is different from the component to which the output is applied (see Fig. 3). Just like luma and chroma ALF filtering, CCALF supports DMS filtering, which helps to reduce the required number of coefficients and the number of multiply and accumulate operations required to achieve its implementation. In the initial VVC proposal, the filter size was a 5×6 diamond shape, but was further reduced to a 3×4 diamond shape in the final version of the VVC standard [25].

4 Versatile video decoder (VVdeC)

The Fraunhofer Heinrich Hertz Institute released the first version of an optimised VVC decoder, named Versatile Video Decoder (VVdeC), on 6 October 2020 [29]. The objective of this decoder is to have a real-time implementation of the VVC standard optimised for different platforms. It is an open-source VVC decoder based on VVC test model (VTM) software, supporting the VVC Main 10 profile. VVdeC fully supports the decoding of all bit streams encoded using the VVC standard. Furthermore, it is compatible with FFmpeg [30] and GPAC [31].

4.1 Parallel implementation of VVdeC

To attain the aimed performance, VVdeC exploits multithreading and SIMD parallelisation. VVdeC starts by parsing multiple frames. Then, the reconstruction process is applied on the parsed frames by splitting the tasks into CTU and CTU line based. For tracking among tasks, a stage is given to each CTU. It helps to execute tasks simultaneously when the dependencies are settled. In this process, a task worker is allocated to each CTU and task workers are given available tasks through scanning by thread pool. The decoding of a frame is completed after the filtering process for all CTUs is completed. Compared to VTM, VVdeC has shown a reduction of up to 90% [8] of the decoding time.

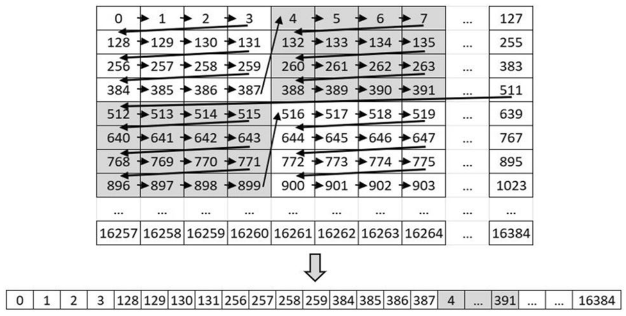

Fig. 4 Conversion of data access pattern

4.2 ALF in VVdeC

To compute ALF, the assignment of tasks to threads in VVdeC considers a main thread that launches and controls the working threads. Each working thread computes ALF for each CTU that the main thread assigns, provided that the SAO filtering of neighbouring CTUs has been completed. This guarantees that the necessary preconditions are satisfied for the ALF filtering. The number of CTUs assigned to a working thread increases with the number of CTUs per row of the picture, but decreases with the number of working threads.

5 Proposed GPU implementation of VVdeC ALF

In our previous work [32], VVdeC (version 1.3) was ported to the ARM instruction set architecture (ISA), because it was originally designed for the x86-based architecture [8, 33]. The migration process involved removing dependencies, including external libraries, and deleting/adapting some formal optimisation designed for x86-based architecture. Finally, VVdeC was optimised using ARM Neon SIMD to maximally exploit data-level parallelism on the CPU side. The resulting source code is openly available [34]. This work extends our previous work with a GPU-based implementation of the VVdeC ALF block on a heterogeneous platform using the Compute Unified Device Architecture (CUDA) programming API. It comprises the following aspects considered on the migration process: programme redesign, data ordering, memory allocation, data transfer, kernel distribution, and task schedule.

5.1 Program redesigning

The GPU-based implementation of VVdeC ALF started by redesigning the CPU-based programme to more efficiently support parallelisation using the GPU. In VVdeC, the ALF

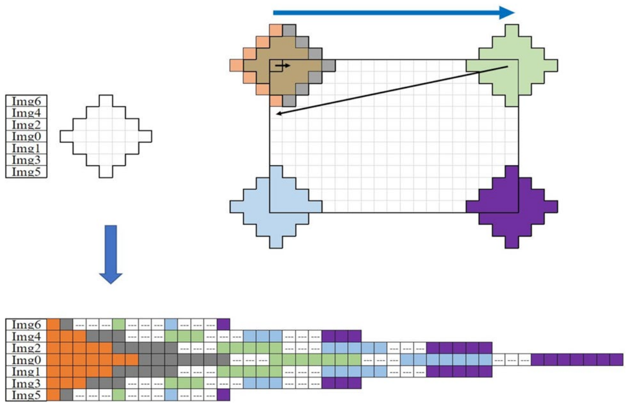

Fig. 5 7×7 diamond-shaped (top left), sliding diamond-shaped filter over CTU (top right), and data ordering pattern in the first approach (bottom)

7×7 and 5×5 DMS filters consist of four nested *for* loops that are placed one inside another. Here, the two inside loops go through the block of 4×4 pixels that is being filtered, while the other two go through the CTU with a stride of 4 pixels for each iteration. The data access pattern of VVdeC ALF is illustrated in Fig. 4 (top) for a CTU of 128×128 pixels.

To achieve a more efficient implementation in the GPU, the four nested *for* loops in the source code were replaced with a single *for* loop that iterates over all pixels of the CTU. The conversion of the data access pattern is shown in Fig. 4 (bottom). Afterwards, moving to the GPU was straightforward, since the size of the *for* loop was replaced with the number of GPU threads that are launched. In addition, each thread filters a single pixel in the same way as is done on the CPU. However, the data address pointers are modified, as explained in Sect. 5.2. In total, 128 threads per block have been used to exploit maximum GPU load and performance. The number of blocks is the number of pixels to be filtered divided by 128 (threads per block).

5.2 Data ordering

The 7×7 DMS filter requires up to three pixels above, on the left, right, and bottom, following a diamond shape, as illustrated in Fig. 5 (top left) and (top right). For 5×5 DMS filtering, the processing is the same as for 7×7 DMS filtering, but it only needs up to two pixels.

Three distinct approaches were considered to implement the filter in the GPU. In a first approach, all pixels within the diamond shape are copied into different variables depending on their row (Img) in the diamond, as shown in Fig. 5

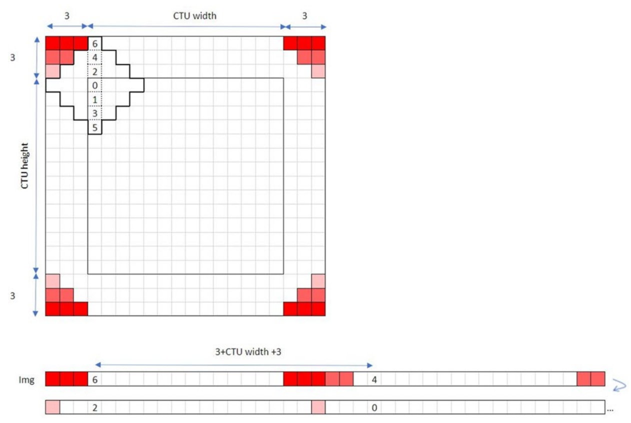

Fig. 6 Data ordering pattern in the final approach

Table 1 Reduction of data copied on the basis of different data ordering approaches

CTU size	Transferred data (Bytes)			Reduction over App. 1	
	App. 1 (B)	App. 2 (B)	App3 (B)	App. 2 (%)	App. 3 (%)
128x128	409600	116992	17956	71.4%	95.6%
64×64	102400	29824	4900	70.8%	95.2%

(bottom). Here, the same pixel information is copied several times to filter adjacent pixels.

The second approach aimed to reduce duplicated pixels copied to the GPU. For each CTU row, all pixels in each ImgX row are copied, where X = 0-6. Therefore, horizontally adjacent pixels can share the same pixel data, but vertically adjacent pixels use different data. In this case, the same pixels are duplicated vertically to filter the vertical adjacent pixels. Compared to the first approach, this one requires less data copy, and the implementation complexity is simpler. However, the same pixels are still copied more than once.

In the final approach, the CTU is copied along with three pixels (in total six per corner) in each direction: above, left, right, and bottom, as shown in Fig. 6 (top). These extra pixels (in total six per corner) around the CTU are copied, since those are also needed for filtering the pixels on the border of the CTU. Therefore, the bidimensional CTU is transformed into a one-dimensional array by concatenating each row of the original CTU, as shown in Fig. 6 (bottom). Here, the pixels at the corners marked with red are not used for both 7×7 and 5×5 DMS filters, but they are also copied to simplify the addressing of the pixels in the GPU code. The advantages of this strategy are the following: (1) only one memory vector holds all pixel information, (2) each pixel is only copied once (the minimum possible), and (3) the implementation complexity is very low. As a disadvantage, the access is not coalescent if a thread wants to get the pixels of different rows (there is a stride of 6 + CTU width), but the reduction of copied data improves more than the coalesced access.

Table 1 shows the amount of data transferred in bytes (B) for the three considered approaches (App.), the two different CTU sizes, and the reduction in the amount of copied bytes (in %) of approaches 2 and 3 compared to the first one (1).

5.3 Memory allocation

The GPU is a coprocessing unit of the CPU that executes the tasks assigned by the CPU. However, both GPU and CPU use different memory address spaces, where the GPU cannot access the CPU memory directly. Therefore, the main bottleneck for CPU+GPU implementation comes from data transfers between CPU and GPU [35]. To optimise the performance of the CPU+GPU implementation, memory allocation needs to be efficiently managed. In this part of the study, different CUDA API functions are discussed to overcome this limitation.

The function *cudaMallocHost* allocates page-locked memory to the host. However, this approach introduces some limitations: When a significant amount of data are allocated, performance decreases. However, *cudaMallocHost* allocates the memory space to the pinned memory [36], which means that data need to be copied only once from the pinned memory to the GPU. On the contrary, data allocation using the *cudaMallocManaged* function requires two copies: (1) unified memory to pinned memory and (2) pinned memory in GPU. Considering the small amount of data per filter (100 KB), the final implementation makes use of *cudaMallocHost* (instead of others, such as *cudaMallocManaged* or *cudaMalloc*), since the copy time is shorter and the resulting global performance is better.

5.4 Data transfer

Data transfers between the CPU and the GPU were implemented using the *cudaMallocHost* function, which makes it possible to use of the *memcpy* function instead of *cudaMemcpy*. It results in faster data transfers and shorter copy times. The *cudaMemcpyAsync* was also tested to verify whether parallelism between copy and CPU execution could improve the performance. However, *cudaMemcpyAsync* showed to be slower in this case, as it took a setup overhead of around 18 μs each time it was called to copy a variable before the asynchronous copy started. On the contrary, the copy time of all the variables of a filter was less than 10 μs when *memcpy* was used synchronously. Moreover, in such a case, *cudaMemcpyAsync* would be called several times by multiple CPU threads at the same time in VVdeC to copy small chunks of data. Thus, causing undesired bottlenecks. When CPU threads run in parallel with asynchronous copy and execute another cudaMemcpyAsync, such an operation must wait until the previous copy finishes. Hence, it can be concluded that *cudaMemcpyAsync* is only useful when dealing with large blocks of data (100 MB or greater). In our implementation, *cudaMemcpyAsync* did not provide benefit, as there are many variables and relatively few data elements per filter (100 KB). *CudaMemPrefetchAsync* was also tested, but it had a small negative impact on performance, as it takes 3.5 μs each time it is executed.

To conclude, the best performance is obtained when synchronously coping with the *memcpy* function in the GPU memory allocated with *cudaMallocHost*.

5.5 Kernel distribution

This section discusses the kernel distribution for ALF filtering on the GPU. To increase the performance, ALF filtering tasks are assigned to the GPU kernels in different ways to maximise the parallel computation of all the filters for a frame.

The ALF filtering process requires reconstructed samples from the SAO filtering process. At the beginning of the ALF filtering on GPU, SAO reconstructed samples are provided to GPU from CPU using *cudaMallocHost*. Therefore, a suitable filter among 7×7 DMS filter (for the luma component) and two 5×5 DMS filters (for the chroma Cb and Cr) is applied to each pixel of the CTU. Moreover, all CTUs are processed simultaneously, as ALF CTUs are independent of each other.

Initially, each 7×7 DMS filter and two 5× 5 DMS filters from each CTU are included in one kernel. This reduced the number of kernels launched by 3× (if the luma and chroma filters are always performed). This approach improves the performance as the kernel requires some time to start before any computation is performed on the GPU. Moreover, this time increases with the number of CPU threads that call a kernel. For example, with 8 threads, on average, 180 μs are consumed for initialisation, while the filtering computation in GPU only takes 7 μs for 7x7 DMS filter. However, the GPU was mostly running continuously. The former observation motivated another implementation: instead of grouping three filters in one kernel, all filters are grouped in a single kernel to process a whole frame. To do that, all the required data are first copied to the memory space allocated with *cudaMallocHost*. After the filter computation at the GPU, a thread sets the frame as reconstructed and ready to be used by other decoder blocks, as all CTUs can be computed simultaneously using the same kernel. Using this approach, about 500 7×7 and 1000 5×5 DMS filters can be executed using a single kernel to decode 3840×2160 UHD sequences. Subsequently, the results of the ALF cross-component functions of the Cb and Cr components are copied from the CPU to the GPU and added with the results obtained from the 5×5 DMS filtering of the Cb and Cr components. Then, the executions on GPU are completed by clipping the results of addition between the cross-component and 5×5 DMS filters, as shown in Fig. 7. Finally, synchronisation is performed to retrieve the results and send them back to the CPU.

By default, only one kernel can run at the same time, so if another is launched (by the same CPU thread or a different one), it waits until the existing one finishes. To solve such an ineffectiveness, the CUDA stream feature is used. It enables the execution of different kernels at the same time by the same application. Thus, different threads can launch kernels that run concurrently, and CPU threads are not blocked, allowing their parallel execution.

5.6 Task schedule

Parallelism between CPU and GPU computations must be optimised to maximise the decoder performance. This requires properly scheduling not only the GPU computations, but also the data transfers between the CPU and the GPU. The double buffering technique was used to deal with this challenge.

In this study, the GPU kernel is launched with all filters in a frame before the frame is set as finished, while the CPU

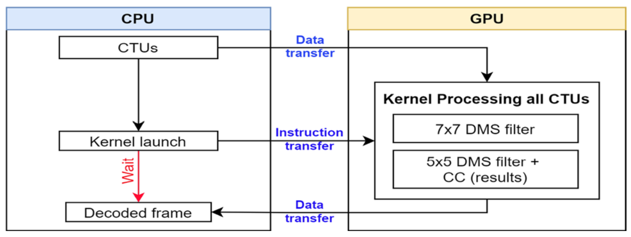

Fig. 7 Diagram of the hybrid approach using CPU+GPU

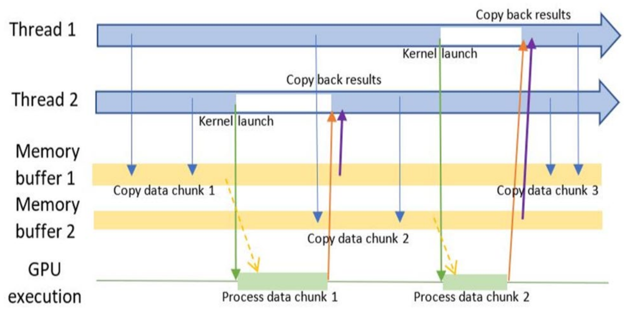

Fig. 8 Diagram of the GPU task scheduling

thread is kept waiting until the GPU finishes. Afterwards, the results are copied to the CPU and the kernel continues its normal execution, setting the frame as finished. Note that this approach does not provide full CPU–GPU parallelism, since other CPU threads can run meanwhile. To address this challenge, two data buffers allocated with *cudaMallocHost* are used, as illustrated in Fig. 8. First, the CPU threads copy the data necessary to compute the filter into one buffer. When a kernel is to be launched, the buffer is changed, so that other CPU threads can still run and continue copying the data in a buffer different from the one being used by the GPU. After the kernel finishes, the CPU thread that launched it can continue to copy the results back and set the image as reconstructed. This buffer change is repeated each time a kernel is launched. In addition, the design has been improved by automatically expanding the buffers when they run out of space, so that more data from the filters can be stored. Such double-buffer implementation guarantees efficient data-to-memory allocation.

To achieve full parallelism, the synchronisation between the CPU and the GPU is performed, and first, the CPU threads fill the first memory buffer. Next, when a thread reaches the end of a frame, the buffer is changed to be filled by other threads. Thus, other threads can continue filling the memory that is not going to be used by the GPU. Meanwhile, the GPU kernel is running, and other threads can continue filling the new buffer. Later, when the kernel finishes, the thread that launched it copies back the results that were stored in the old buffer used by the GPU and continues its execution by setting the frame as finished.

6 Experimental results

The proposed implementation was evaluated using an NVIDIA Jetson AGX Xavier development kit. This is an embedded heterogeneous platform consisting of an 8 core ARM 8.2 64 bit CPU and a 512 core Volta GPU [37]. This platform is equipped with 32 GB of RAM with a transfer rate of 137 GB/s. The CPU has a maximum clock rate of 2.26 GHz, and contains a 8 MB L2 cache memory and a 4 MB L3 cache memory. The GPU has a maximum clock rate of 1.37 GHz and contains a 512 KB L2 cache memory. The embedded platform was configured with the Ubuntu 18.04 operating system, running CMake version 3.16, gcc version 7.5, CUDA version 10.2, and activating the -O3 optimisation. All experiments were carried out using the maximum clock rate of the CPU and GPU.

6.1 Test bench description

In this evaluation, 11 video sequences have been used from common JVET test sequences [38]: 3 Class A1 sequences with resolution 3840×2160 Tango2 (TG2), FoodMarket4 (FM4), and Campfire (CFR); 3 Class A2 sequences with resolution 3840×2160 CatRobot1 (CR1), DaylightRoad2 (DR2), and ParkRunning3 (PR3); and 5 Class B sequences with resolution 1920×1080 MarketPlace (MPL), RitualDance (RUD), Cactus (CCT), BasketballDrive (BBD), and BQTerrace (BQT). These sequences have been encoded by setting the bit depth to 10 for all intra (AI), random access (RA), and low delay (LD) configurations with quantisation parameters (QP) equal to 22, 27, 32, and 37 (see Table 2).

Table 2 Features of the VVC test sequences

Class	Sequence	Resolution	Frames	Bit depth
A1	TG2	3840×2160	294	10
	FM4	3840×2160	300	10
	CFR	3840×2160	300	10
A2	CR1	3840×2160	300	10
	DR2	3840×2160	300	10
	PR3	3840×2160	300	10
B	MPL	1920×1080	600	10
	RUD	1920×1080	600	10
	CCT	1920×1080	500	10
	BBD	1920×1080	500	10
	BQT	1920×1080	600	10

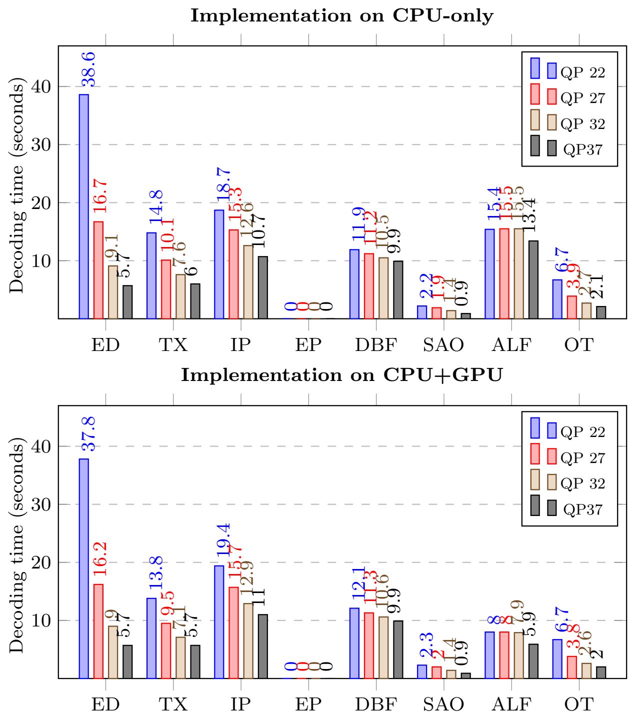

Fig. 9 Average time distribution for different blocks of the VVdeC decoder (in sec.) over CPU-only (top) and over CPU+GPU (bottom) with SIMD activated for all intra configurations

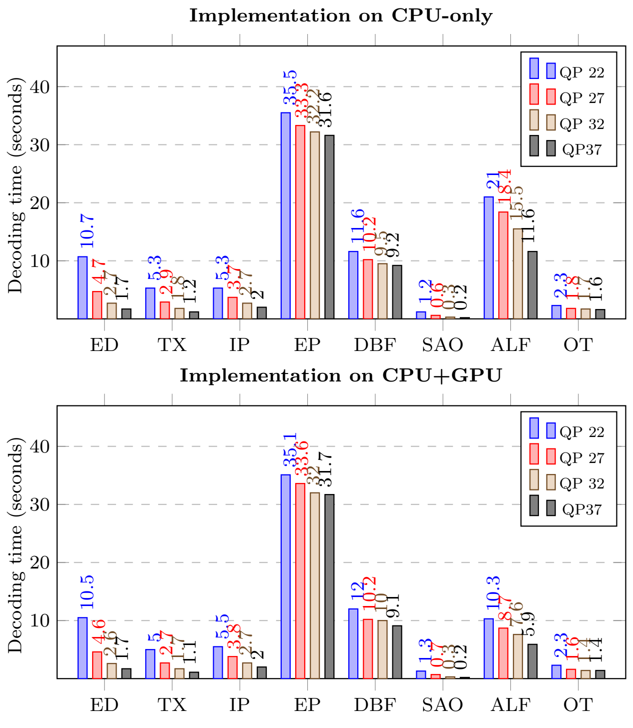

Fig. 10 Average time distribution for VVdeC decoder blocks (in sec.) over CPU-only (top) and over CPU+GPU (bottom) with SIMD activated for random access configurations

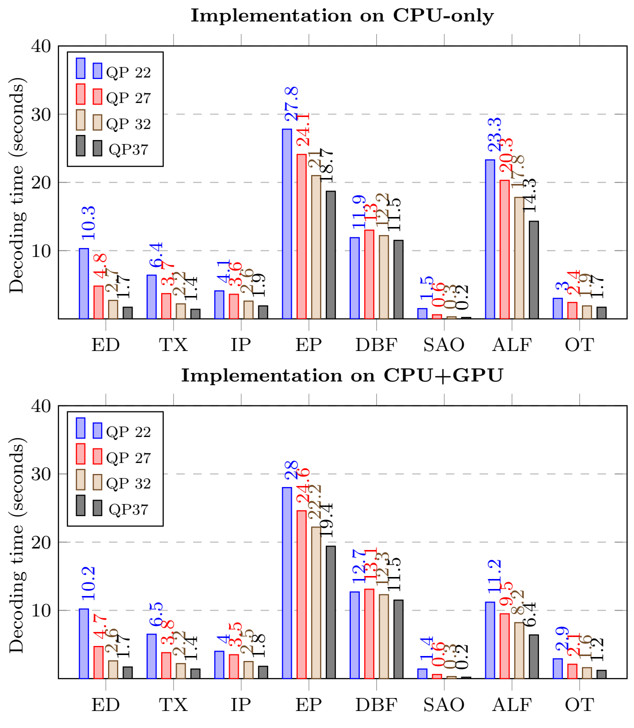

Fig. 11 Average time distribution for VVdeC decoder blocks (in sec.) over CPU-only (top) and over CPU+GPU (bottom) with SIMD activated for LD configurations

6.2 Performance analysis

As was described in the previous section, the conceived CPU+GPU implementation accelerates the ALF block on the GPU, leaving the rest of the decoder on the CPU. Figure 9 shows the average decoding time distribution (in seconds) for the 11 test sequences considered for different VVdeC blocks (in seconds) on the reference CPU-only implementation described in Subsection 5 (top) and on the proposed CPU+GPU implementation (bottom), both with ARM Neon SIMD activated for AI configurations with QP 22-37. As can be observed, all the VVdeC blocks took a similar time on the CPU-only implementation compared to the time taken by the CPU+GPU implementation. The only exception is the ALF block, which obtained an average speedup of 2 in the CPU+GPU implementation. Moreover, the time consumption by ALF on the CPU + GPU implementation was lower than the DBF time by at least 2.7 s. In addition, the computation time of the ED and TX blocks is slightly lower in the CPU+GPU implementation, because they benefited from more CPU availability in the implementation. The conclusions are similar for RA configurations, as presented in Fig. 10. In this case, most of the decoding time was taken by EP, ALF, and DBF on both implementations. ALF consumed (on average) 16.6 s for the CPU-only implementation and 8.1 s for the CPU+GPU implementation. The scenario is also similar for low-delay sequences, where ALF consumed on average 18.9 s for the CPU-only implementation and 8.8 s for the CPU+GPU implementation (see Fig. 11).

Table 3 presents the average speedups of ALF and the entire decoder using the CPU + GPU implementation over the CPU-only implementation for the AI, RA, and LD configurations with SIMD activated.

Table 3 Average speedup obtained for ALF and total decoding time (TOT) using CPU+GPU over CPU-only with SIMD activated

	All intra		Random access		Low delay	
QP	ALF	TOT	ALF	TOT	ALF	TOT
22	1.91	1.08	2.02	1.13	2.07	1.15
27	1.93	1.12	2.09	1.14	2.15	1.16
32	1.95	1.15	2.01	1.13	2.18	1.17
37	2.28	1.18	1.93	1.11	2.22	1.18
Avg	2.02	1.13	2.01	1.13	2.16	1.17

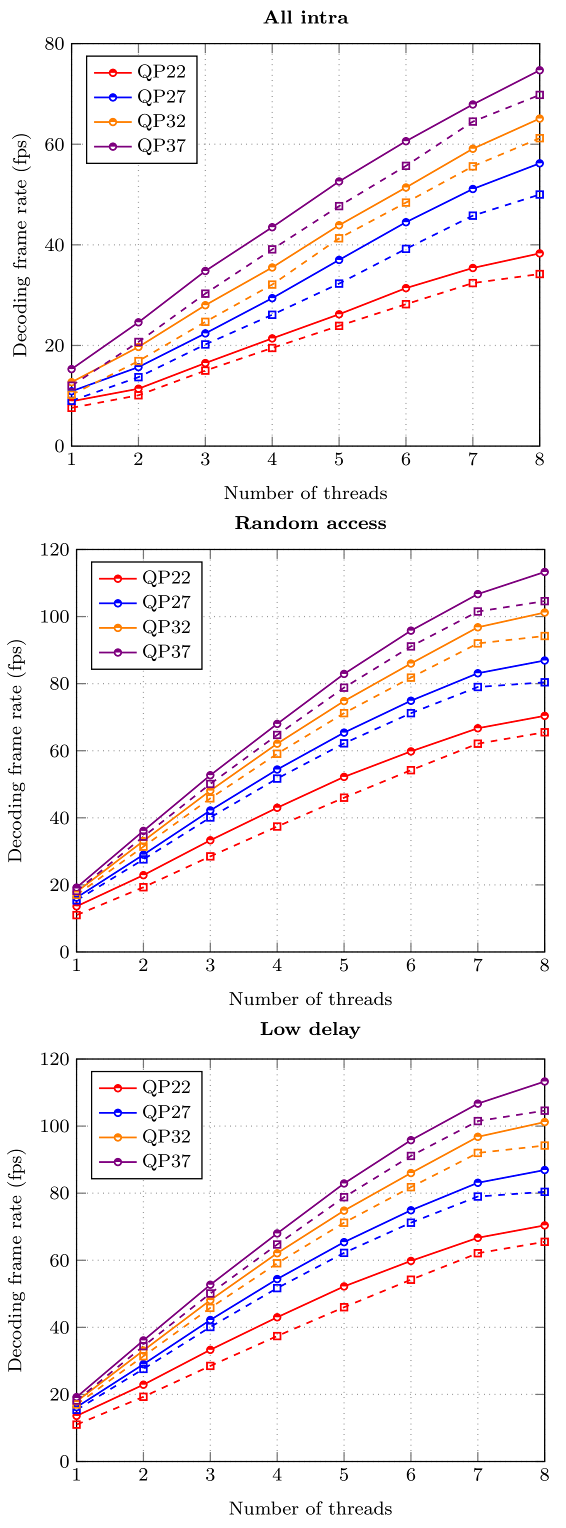

Fig. 12 Average fps obtained for the proposed implementation on (1) CPU-only (dashed line) and (2) CPU + GPU (solid line) with SIMD activated for different thread numbers with QPs 22, 27, 32 and 37 of AI (top), RA (middle), and LD (bottom) FHD sequences

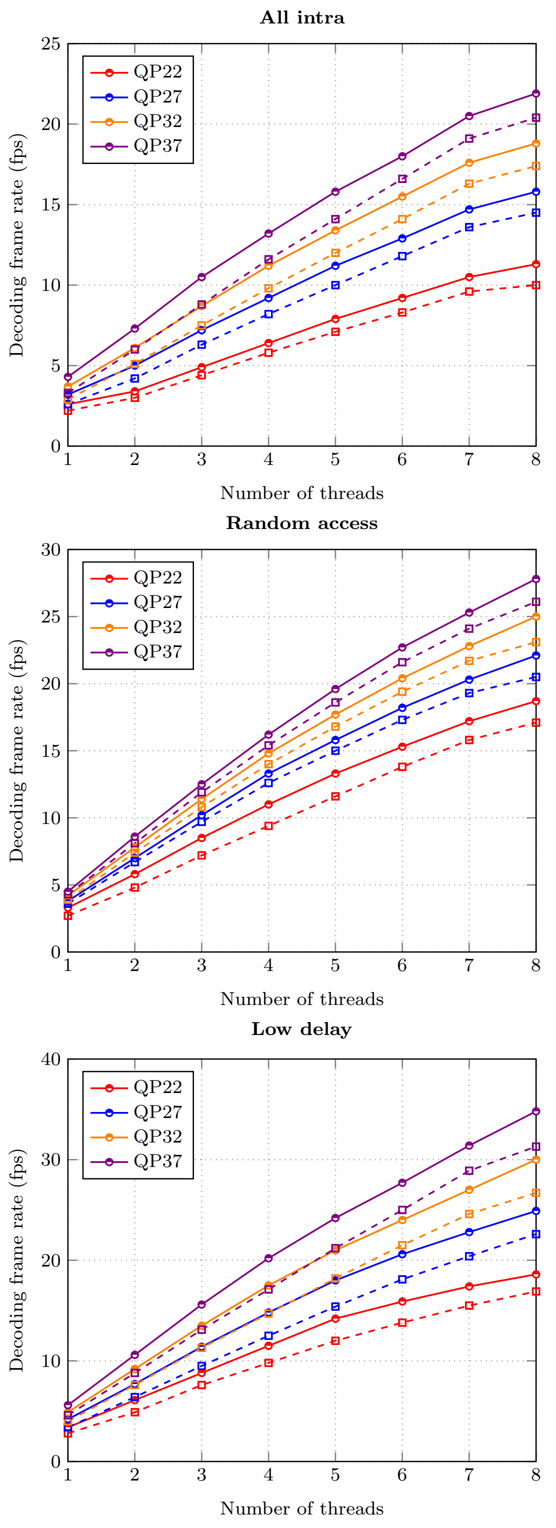

Fig. 13 Average fps obtained for the proposed implementation on (1) CPU-only (dashed line), and (2) CPU+GPU (solid line) with SIMD activated for different thread numbers with QPs 22, 27, 32, and 37 of AI (top), RA (middle), and LD (bottom) UHD sequences

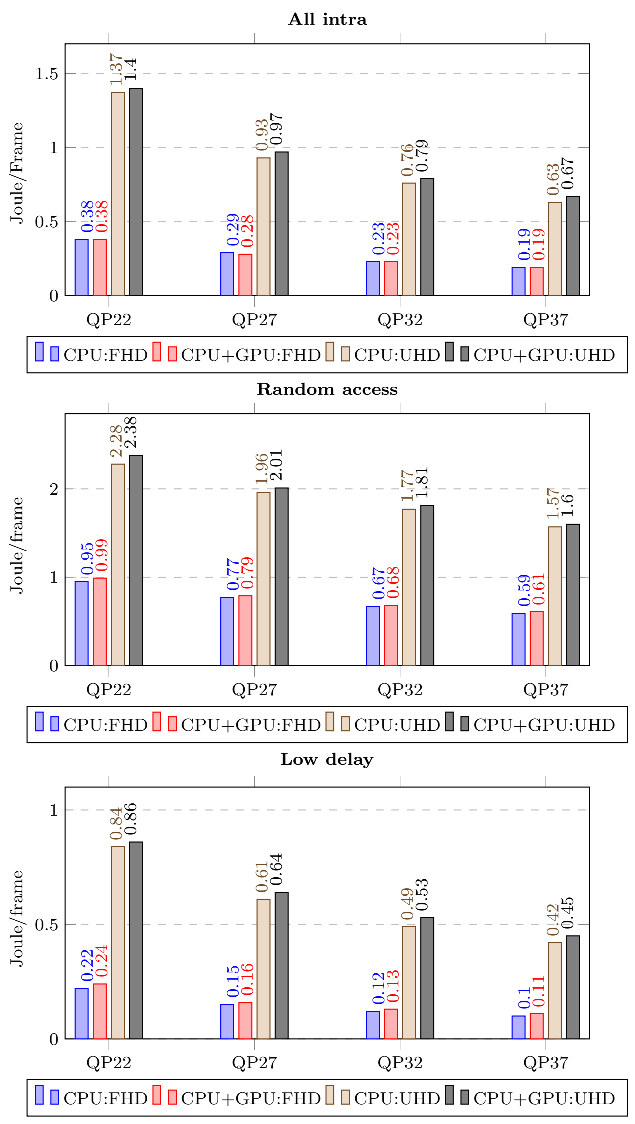

Fig. 14 Average energy consumed (in J per frame) for FHD and UHD over CPU-only and CPU+GPU for AI (top), RA (middle), and LD (bottom) configurations with QPs 22-37 and SIMD activated

Figures 12 and 13 present the average performance in frames per second (fps) of the CPU-only and CPU+GPU implementations with SIMD activated for different executing threads in the CPU using four QP values for AI, RA, and LD configurations UHD and Full High Definition (FHD) sequences, respectively. As can be seen, the CPU+GPU implementation is always faster (at least 1.1 times) than the CPU-only implementation, independently of the amount of used cores. For the RA and LD configurations, the CPU+GPU implementation achieved a speed-up of 1.2 times when compared to the CPU-only implementation using one core. For eight cores, it is up to 1.1 times faster than the CPU-only implementation. On average, ALF consumed around 20% and 12% of the decoding time in the CPU-only and CPU+GPU implementations, respectively. Furthermore, all FHD sequences achieved real-time decoding in the CPU + GPU implementation on a resource-constrained mobile embedded platform using 8 cores, except AI sequences with QP equal to 22. Moreover, the decoding of 4K UHD LD sequences with QP 32-37 was in real time. The average maximum/minimum fps obtained by AI, RA, and LD UHD sequences with eight cores were 21.9/11.9, 27.8/18.7, and 34.8/18.6 fps, respectively.

6.3 Energy consumption

In addition to performance, energy consumption is another factor to consider in resource-constrained embedded platforms. The average power consumption of the platform was measured by reading the power consumption each time a frame was decoded. Subsequently, the average energy consumption (in J) of the entire sequence was calculated by multiplying the average power consumption during the decoding by the decoding time. To measure power, the Jetson AGX Xavier is equipped with integrated sensors to obtain power consumption at the CPU and GPU (in mW) whose values are registered in two files in the filesystem. For CPU-only execution, the GPU energy is not zero, because the GPU is still enabled, while for the CPU+GPU, both consumptions are presented.

In Fig. 14, the average energy consumption per frame (in J/frame) of FHD and UHD sequences on CPU-only and CPU+GPU is presented for the AI (top), RA (middle), and LD (bottom) configurations with QP equal to 22-37 and SIMD activated. As expected, the average energy consumption was similar but slightly higher for the CPU+GPU implementation compared to the CPU-only implementation. Moreover, sequences with higher QP (lower quality and less computational load) consumed less energy than sequences with lower QP (higher quality and more computational load) for all configurations. Furthermore, the FHD sequences consumed 2× to 3× less energy compared to the UHD sequences for all configurations. The maximum/minimum average energy consumption per frame of the CPU-only implementation was 1.37/0.19 J/frame for the AI configuration, 2.28/0.59 J/frame for the RA configuration, and 0.84/0.10 J/frame for the LD configuration. For the CPU+GPU implementation, the maximum/minimum average energy consumption was 1.40/0.19 J/frame, 23.8/0.61 J/frame, and 0.86/0.11 J/frame for the AI, RA, and LD configurations, respectively.

Accordingly, it can be concluded that active use of the GPU implies a slight increase in the energy consumed per frame (2.9%) which is compensated by the provided speedup. This seems fair considering that GPU modules usually involve higher power consumption than standard CPU modules, despite the difference in operating frequency.

However, the CPU does not benefit from the transfer of computational load to the GPU, as it is on a waiting time during these periods. In a commercial implementation, the CPU might be using such clock cycles to perform other tasks simultaneously, thus making a more efficient use of energy resources.

7 Conclusions

This article proposes a hybrid approach to accelerate an optimised versatile video decoder (VVdeC) ALF filter using a GPU. The GPU has been comprehensively used by redesigning the VVdeC ALF programme to maximise the degree of parallelism over resource-constrained heterogeneous embedded platforms. The proposed approach allowed to accelerate ALF computation by an average of two times for AI, RA, and LD video sequences in an NVIDIA AGX Jetson Xavier platform. Furthermore, the proposed CPU+GPU implementation with SIMD activated offers an average rate of 48 fps for AI sequences, 69 fps for RA sequences, and 80 fps for LD sequences. The results obtained also show an average speedup of 1.1 for the total decoding time compared to an already fully optimised version of the software decoder. In addition, this paper presents an analysis of energy consumption, a key factor in the targeted embedded platforms. The CPU+GPU implementation with SIMD activated consumed similar energy compared to the CPU-only implementation with SIMD activated for sequences with different configurations. For future work, other potential VVC blocks could be migrated to GPU. The following aspects are considered as future work to improve the current implementation: (1) the processing of multiple pixels per processing thread, and (2) the development of a strategy to use limited shared memory of the embedded platform used in the research to reduce global memory access. Finally, work is underway to design power consumption models that dynamically adapt, depending on the needs of both the environment and the video performance, to the most advantageous scenario between moving blocks of the algorithm to one or another processor, CPU or GPU.

Author Contributions AS, VA, and TD mainly worked on the implementation, software development, and testing. NR, MC, and FP contributed to idea generation, general work supervision, and looked for funds to support the research work. All authors contributed to the manuscript writing and review.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. This research was funded by Spanish Ministerio de Ciencia y Innovación, under Grant No. TALENT (PID2020-116417RB-C41), and Portuguese national funds through Fundacao para a Ciência e a Tecnologia (FCT) under Grant No. UIDB/50021/2020.

Data availability No data was used for the research described in the article.

Declarations

Conflict of interest The authors declare no competing interests.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Fraunhofer HHI is proud to present the new state-of-the-art in global video coding: H.266/VVC brings video transmission to new speed. https://newsletter.fraunhofer.de/-viewonline2/17386/465/11/14SHcBTt/V44RELLZBp/1 (2022). Accessed 5 Jun (2022)
2. 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)
3. Feldmann, C.: Versatile video coding hits major milestone. https://bitmovin.com/compression-standards-vvc-2020 (2022). Accessed 05 June (2022)
4. Pakdaman, F., Adelimanesh, M.A., Gabbouj, M., Hashemi, M.R.: Complexity analysis of next-generation VVC encoding and decoding. IEEE Int. Conf. Image Process. (ICIP) **2020**, 3134–3138 (2020). https://doi.org/10.1109/ICIP40778.2020.9190983
5. Saha, A., Chavarrías, M., Pescador, F., Groba, Á.M., Chassaigne, K., Cebrián, P.L.: Complexity analysis of a versatile video coding decoder over embedded systems and general purpose processors. Sensors **21**, 3320 (2021). https://doi.org/10.3390/s21103320
6. Franklin, D.: NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics, y. https://developer.nvidia.com/blog/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/ (2022). Accessed 07 June 2022
7. Li, S., Wang, R., Yao, K.: CUDA acceleration for AVS2 loop filtering. IEEE Second Int. Conf. Multimed. Big Data (BigMM) **2016**, 246–250 (2016). https://doi.org/10.1109/BigMM.2016.66
8. Wieckowski, A., et al.: Towards a live software decoder implementation for the upcoming versatile video coding (VVC) codec. IEEE Int. Conf. Image Process. (ICIP) **2020**, 3124–3128 (2020). https://doi.org/10.1109/ICIP40778.2020.9191199
9. Gudumasu, S., Bandyopadhyay, S., He, Y.: Software-based versatile video coding decoder parallelization. Proc. ACM Multimed. Syst. Conf. (2020). https://doi.org/10.1145/3339825.3391871
10. Zhu, B., et al.: A real-time H.266, VVC software decoder. IEEE Int. Conf. Multimed. Expo (ICME) **2021**, 1–6 (2021). https://doi.org/10.1109/ICME51207.2021.9428470

11. Li, Y., et al.: An optimized H266/VVC software decoder on mobile platform. Pict. Coding Symp. (PCS) (2021). https://doi.org/10.1109/PCS50896.2021.9477484
12. Han, X., Wang, S., Ma, S., Gao, W.: Optimization of motion compensation based On GPU and CPU For VVC decoding. IEEE Int. Conf. Image Process. (ICIP) **2020**, 1196–1200 (2020). https://doi.org/10.1109/ICIP40778.2020.9190708
13. Vázquez, M.F., Saha, A., Morillas, R.M., Lapastora M.C., Oso, F.P. D.: Work-in-progress: porting new versatile video coding transforms to a heterogeneous GPU-based technology. In: International Conference on Compliers, Architectures and Synthesis for Embedded Systems, pp. 1–2 (2019)
14. OpenHEVC software repository. https://github.com/OpenHEVC/openHEVC (2022). Accessed 14 May 2022
15. Wang, Y., Guo, X., Fan, X., Lu, Y., Zhao, D., Gao, W.: Parallel in-loop filtering in HEVC encoder on GPU. IEEE Trans. Consum. Electron. **64**(3), 276–284 (2018). https://doi.org/10.1109/TCE.2018.2867812
16. de Souza, D.F., Ilic, A., Roma, N., Sousa, L.: HEVC in-loop filters GPU parallelization in embedded systems. Int. Conf. Embed. Comput. Syst. (2015). https://doi.org/10.1109/SAMOS.2015.7363667
17. de Souza, D.F., Ilic, A., Roma, N., Sousa, L.: GPU-assisted HEVC intra decoder. J. Real-Time Image Proc. **12**(2), 531–547 (2016). https://doi.org/10.1007/s11554-015-0519-1
18. Ma, A., Guo, C.: Parallel acceleration of HEVC decoder based on CPU+GPU heterogeneous platform. Seventh Int. Conf. Inf. Sci. Technol. (ICIST) **2017**, 323–330 (2017). https://doi.org/10.1109/ICIST.2017.7926778
19. Zhang, W., Guo, C.: Design and implementation of parallel algorithms for sample adaptive offset in HEVC based on GPU. Sixth Int. Conf. Inf. Sci. Technol. (ICIST) **2016**, 181–187 (2016). https://doi.org/10.1109/ICIST.2016.7483407
20. Ma, S., Huang, T., Wen, G.: The second generation IEEE, video coding standard. In: IEEE China Summit and International Conferece on Signal and Information Processing, p. 2015 (1857)
21. Jiang, B., Xu, H., Luo, F., Wang, S., Ma, S., Gao, W.: GPU-based intra decompression for 8K real-time AVS3 decoder. IEEE Conf. Multimed. Inf. Process. Retr. (MIPR) (2020). https://doi.org/10.1109/MIPR49039.2020.00061
22. Han, X., et al.: GPU based Real-Time UHD Intra Decoding for AVS3. IEEE Int. Conf. Multimed. Expo Worksh. (ICMEW) **2020**, 1–6 (2020). https://doi.org/10.1109/ICMEW46912.2020.9106009
23. Jiaqi, Z., Chuanmin, J., Meng, L., Shanshe, W., Siwei, M., Wen, J.: Gao. Recent development of AVS video coding standard : Avs3. In: 2019 Picture Coding Symposium (PCS), IEEE, pp. 311–315 (2019)
24. Karwowski, D.: Precise probability estimation of symbols in VVC CABAC entropy encoder. IEEE Access **9**, 65361–65368 (2021). https://doi.org/10.1109/ACCESS.2021.3075875
25. Karczewicz, M., et al.: VVC in-loop filters. IEEE Trans. Circuits Syst. Video Technol. **31**(10), 3907–3925 (2021). https://doi.org/10.1109/TCSVT.2021.3072297
26. Tsai, C.-Y., et al.: Adaptive loop filtering for video coding. IEEE J. Sel. Top. Signal Process. **7**(6), 934–945 (2013). https://doi.org/10.1109/JSTSP.2013.2271974
27. Erfurt, J., et al.: Extended multiple feature-based classifications for adaptive loop filtering. APSIPA Trans. Signal Inf. Process. **8**, 28 (2019). https://doi.org/10.1017/ATSIP.2019.19
28. Wang, X., Sun, H., Katto, J., Fan, Y.: A hardware architecture for adaptive loop filter in VVC decoder. IEEE 14 Int. Conf. ASIC (ASICON) **2021**, 1–4 (2021). https://doi.org/10.1109/ASICON52560.2021.9620332
29. Fraunhofer HHI VVdeC Software Repository. https://github.com/fraunhoferhhi/vvdec (2022). Accessed 02 July 2022
30. Ffmpeg.: A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/ (2022). Accessed 22 Aug 2022
31. GPAC: Multimedia Open Source Project. https://gpac.wp.imt.fr/ (2022). Accessed 22 Aug 2022
32. Saha, A., Chavarrías, M., Aranda, V., Garrido, M.J., Pescador, F.: Implementation of a real-time versatile video coding decoder based on VVdeC over an embedded multi-core platform. IEEE Trans. Consum. Electron. (2022). https://doi.org/10.1109/TCE.2022.3202512
33. Fraunhofer HHI VVdeC software repository, Releases vvdec-0.2.0.0. https://github.com/fraunhoferhhi/vvdec/releases/tag/v0.2.0.0 (2022). Accessed 27 Apr 2022
34. Saha, A.: VVdeC2_ARM_Neon. https://github.com/Sahamec/VVdeC2_ARM_Neon (2022). Accessed 19 May 2022
35. Sunitha, N.V., Raju, K., Chiplunkar, N.N.: Performance improvement of CUDA applications by reducing CPU-GPU data transfer overhead. Int. Conf. Invent. Commun. Comput. Technol. (ICICCT) (2017). https://doi.org/10.1109/ICICCT.2017.7975190
36. Ponnuraj, R.P.: CUDA memory model. https://medium.com/analytics-vidhya/cuda-memory-model-823f02cef0bf (2022). Accessed 12 Aug 2022
37. NVIDIA Jetson AGX Xavier Developer Kit, User Guide. DA_09403_003, December 17, 2019. https://developer.nvidia.com/jetson-agx-xavier-developer-kit-user-guide (2022). Accessed 06 July 2022
38. Bossen, F., Boyce, J., Li, X., Seregin, V., Sühring, K.: JVET common test conditions and software reference configurations for SDR Video, Document JVET-N1010. JVET of ITU-T, Geneva (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.