GPU Acceleration of the HEVC Decoder Inter Prediction Module

Diego F. de Souza, Aleksandar Ilic, Nuno Roma and Leonel Sousa
INESC-ID, IST, Universidade de Lisboa
Rua Alves Redol 9, 1000-029, Lisbon, Portugal
Email: {diego.souza,aleksandar.ilic,nuno.roma,leonel.sousa}@inesc-id.pt

***Abstract*—The inter prediction decoding is one of the most time consuming modules in modern video decoders, which may significantly limit their real-time capabilities. To circumvent this issue, an efficient acceleration of the HEVC inter prediction decoding module is proposed, by offloading the involved workload to GPU devices. The proposed approach aims at efficiently exploiting the GPU resources by carefully managing the processing within the computational kernels, as well as by optimizing the usage of the complex GPU memory hierarchy. The obtained experimental results show that real-time video decoding is achieved for all tested Ultra HD 4K, WQXGA and Full HD video sequences, even when considering the most demanding encoding parameterizations, delivering average processing times up to 20.39 ms, 9.01 ms and 5.22 ms, respectively.**

I. Introduction

The High Efficiency Video Coding (HEVC) encoders have proven to provide equivalent subjective visual quality, while achieving an average bit rate reduction of 50%, when compared with the previous standards (e.g., H.264/MPEG-4 AVC) [1]. However, such coding efficiency comes at the cost of a substantial increase of the computational complexity of both the video encoder and decoder. In what concerns the decoder subsystem, the Inter Prediction Decoding (IPD) module is responsible for 43-49% of the total decoding time in both ARM and x86 instruction set architectures [2]. This is mainly due to the significant set of different block sizes that has to be considered and to the involved pixel interpolation procedures [3], which required a high memory bandwidth and number of arithmetic operations.

To provide the fully compliant HEVC real-time encoding/decoding, current research trends aim at accelerating the execution of particular modules by offloading their computations from the Central Processing Unit (CPU) to different co-processors/accelerators. The majority of these works specifically focuses on exploiting the processing capabilities of nowadays Graphics Processing Units (GPUs), mainly due to their widespread availability in many high performance platforms, as well as in desktop and embedded systems. When considering only the encoder side, the existing GPU-based implementations mainly deal with the computationally demanding motion estimation, as proposed in [4] and [5] for HEVC, and in [6] for H.264/MPEG-4 AVC. However, parallel implementations also pose difficult challenges at the decoder side, mainly because the decoder should be able to decode bitstreams produced by any encoder configuration.

To circumvent the involved computational effort, Chi et al. [7] extensively exploited the usage of Single Instruction, Multiple Data (SIMD) techniques to implement the HEVC decoder modules, by specifically focusing on modern multi-core CPU architectures. In particular, the highest performance in the Intel Haswell architecture was achieved with the Advanced Vector Extensions 2 (AVX2), being the IPD module 10.33× faster than its scalar version. To further increase the attained performance, these authors also divide the computational load among the several CPU cores, by relying on an alternative method based on the HEVC Wavefront Parallel Processing (WPP) [8], thus achieving 543 frames per second (fps) for Full HD video sequences (on average) with an 8-core CPU.

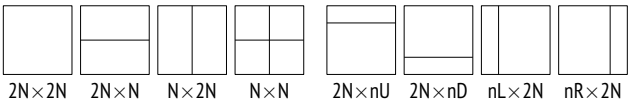
In what concerns GPU implementations, Wang et al. [9] presented kernel designs of the H.264/MPEG-4 AVC interpolation module on OpenCL, aiming a reduction of the performance penalties imposed by the control and memory divergences. Nevertheless, despite the absence of existing approaches that tackle GPU implementations of the entire HEVC decoder or even only the IPD module, several other individual decoding modules have already been proposed by the authors of this paper targeting high performance GPU platforms [10]–[13] and embedded GPUs [14].

In accordance, a new GPU parallel implementation of the IPD module is herein proposed. To the best of the authors' knowledge, the presented IPD parallel implementation represents one of the first approaches to handle this HEVC decoding module in state-of-the-art GPUs. As a result, the proposed algorithm allows achieving processing times as low as 20.4 ms for Ultra HD 4K frames on Compute Unified Device Architecture (CUDA) capable GPUs. The CUDA was chosen instead of OpenCL, due to the possibility of fining tune the GPU, e.g., Shared/L1 memory space configurations.

This paper is organized as follows: the HEVC IPD is summarized in Section II and the proposed algorithm is presented in Section III, while the experimental results and conclusions are addressed in Sections IV and V, respectively.

II. HEVC Inter Prediction

Similarly to the previous video standards, the IPD techniques adopted by HEVC aim to predict a pixel block by using information from temporal neighboring frames, also known as reference frames. Those reference frames are stored in two picture buffers, i.e., List 0 and List 1.



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

Fig. 1. PU partition modes for the HEVC inter prediction.

On the decoder side, the IPD is executed according to the motion data encoded in the received bitstream, including: *i*) the pixel block size; *ii*) prediction direction, which defines the used picture buffers (List 0, List 1 or both); *iii*) reference frame indexes, which specify the frames used in each list; and *iv*) motion vectors, which define the displacement between the positions of the original block and its predictions in the frames.

A. Block Partitioning Structure

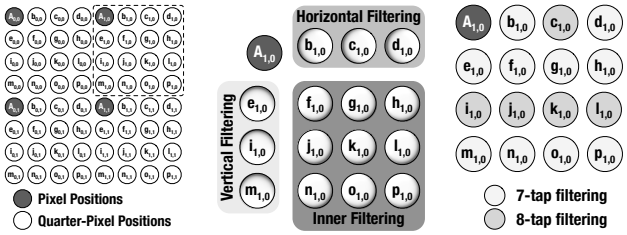
Each video sequence frame is partitioned in $L \times L$ pixel blocks, denoted as Coding Tree Units (CTUs), where the size of each CTU is selected by the encoder ($L \in \{16, 32, 64\}$). Each CTU is then independently split using a quadtree structure in blocks denoted as Coding Units (CUs), between a maximum size of 64×64 and a minimum size of 8×8 pixels, according to a set of criteria. Finally, each CU is further divided in a Prediction Unit (PU) and a Transform Unit, corresponding to the predicted and the residual blocks, respectively [15].

The same frame partitioning (CTU, CU and PU) is applied to each component, i.e., luma and both chromas. Actually, the PU is further divided in luma and chroma Prediction Blocks (PBs), where the IPD is applied to each PB. In particular, when the usual 4:2:0 chroma subsampling is adopted, the chroma blocks are four times smaller than the corresponding luma blocks. Further, when a CU is encoded using Inter prediction, the corresponding PU is split into one, two or four PUs. In Fig. 1, all possible PU partition modes that are allowed by the HEVC standard are shown for the inter-coded CU and grouped in two subsets, i.e., *symmetric* and *asymmetric*.

For a $2N \times 2N$ CU, the *symmetric* partitioning is restricted to the quadtree structure, where a PU is split in up to four blocks (see Fig. 1a). However, the PU can be divided in four blocks only if the CU could not be split into four CUs and the CU size is greater than 8×8 luma pixels. Moreover, the HEVC standard also introduced *asymmetric* partition modes for Inter prediction (see Fig. 1b), which allow more accurate predictions and offer up to 2.8% of bit-rate reduction [16]. Nevertheless, the *asymmetric* partition modes are unavailable when the CU size is equal to the minimum allowed size, in order to reduce the computational load. In this manner, for an 8×8 CU, the possible PU partitions are 8×8, 8×4 and 4×8.

B. Block Inter Prediction

At the decoder, whenever the IPD is performed within a single picture buffer (i.e., List 0 or List 1), the pixel samples of the PB are obtained by fetching a pixel block from the specified reference frame and picture buffer. The position of the pixel block is defined in the motion vector, with its horizontal (x) and vertical (y) components. When the motion vector points to a position of the pixel (see $A_{x,y}$ in Fig. 2a), the PB samples are directly obtained from the reference frame, i.e., no interpolation is performed. Otherwise, when the motion vector indicates a sub-pixel position, an interpolation procedure is started to obtain the fractional samples at positions from $b_{x,y}$ to $p_{x,y}$ in Fig. 2a [17].



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

Fig. 2. Luma sample positions at quarter-pel resolution and filtering features.

As the H.264/MPEG-4 AVC, the HEVC standard also specifies motion vectors at luma quarter-pixel resolution, but with different interpolation procedure. To generate the luma sub-pixel samples, the HEVC standard defines three filtering types: *Horizontal*, *Vertical*, and *Inner Filtering* (see Fig. 2b). In the *Horizontal Filtering*, $b_{x,y}$, $c_{x,y}$ and $d_{x,y}$ samples are computed by filtering the pixels from the same row. In the *Vertical Filtering*, $e_{x,y}$, $i_{x,y}$ and $m_{x,y}$ samples are computed by considering the pixels in the same column of the reference frame. The samples produced by the *Inner Filtering* (see Fig. 2b) are obtained by performing the vertical filtering on the samples from the same column, i.e., the previously produced sub-pixels $b_{x,y}$, $c_{x,y}$ or $d_{x,y}$ with *Horizontal Filtering*. For example, the *Inner Filtering* of $f_{x,y}$, $j_{x,y}$ or $n_{x,y}$ is performed by using $b_{x,y}$ samples. Hence, in *Inner Filtering*, the corresponding sub-pixel samples should be generated first with *Horizontal Filtering* and, only after, the vertical filtering should be applied.

For the luma component, the interpolation is implemented by adopting 8-tap and 7-tap filters, according to each sub-pixel position. The 7-tap filtering is applied to create the sub-pixel samples that are close to the pixels, i.e., light gray filled sub-samples in Fig. 2c, while the remaining sub-samples are produced with 8-tap filtering. In what concerns the chroma interpolation, the filtering is similar as for the luma component, but only 4-tap filters are used, where sub-samples at units $1/8$ of the distance between chroma pixels can be generated.

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

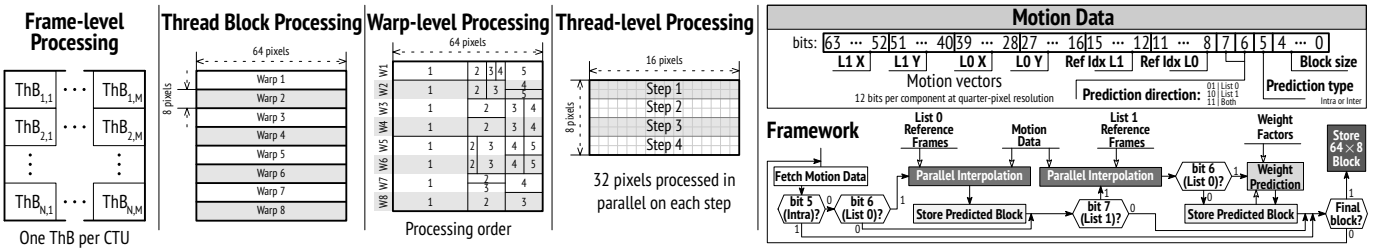


Fig. 3. GPU inter prediction warps assignment and framework.

III. Proposed Inter Prediction Decoding Parallelization

The IPD algorithm proposed herein leverages the fine-grain parallelism of this computationally complex module, while providing fully standard compliant HEVC decoding. The GPU execution is organized in groups of 32 parallel threads (warps), which are grouped in several Thread Blocks (ThBs). To increase the performance, the proposed algorithm maximizes the number of active warps, while ensuring that all threads in a warp perform the same operation from the GPU code (kernel). Furthermore, the data accesses are carefully managed to efficiently exploit the complex GPU memory hierarchy, i.e., global, cache, shared and constant memory.

As it is shown in Fig. 3 (see *Frame-level* and *Thread Block Processing*), a single ThB composed of eight warps is assigned to process each 64×64 luma pixels. Hence, each warp predicts a 64×8 pixel luma sub-block and its corresponding chroma sub-block. If a $N \times N$ PU is larger than eight pixels in the vertical axis, each warp Wi will perform the prediction of its $N \times 8$ sub-blocks (see *Warp-level Processing* in Fig. 3). Each pixel in a sub-block is predicted by one thread of the warp, where 32 pixels are predicted in each step, e.g., a 16×8 sub-block is predicted in four steps (see *Thread-level Processing* in Fig. 3).

To predict each individual sub-block, the required motion data is packed into a 64-bit word (see *Motion Data* in Fig. 3). The first five bits (*Block Size*) represent all 23 allowed PU partitioning sizes $N \times M$, where N and M can be 64, 48, 32, 24, 16, 12, 8 and 4. The bit 5 refers to the block *Prediction Type* (i.e., Intra or Inter), while bits 6 and 7 specify the *Prediction Direction*. The two subsequent sets of 4 bits define the reference frame indexes (*Ref Idx*) for List 0 (*L0*) and List 1 (*L1*). The following four sets of 12 bits are allocated to store the motion vectors at quarter-pixel resolution in each axis, i.e., *X* and *Y*, for each List. Accordingly, the maximum allowed range for a motion vector in a given direction is from -512 to 511 at integer pixel resolution, or from -2048 to 2047 at quarter-pixel resolution. To further reduce the communication overhead, only two 64-bit word *Motion Data* per 8×8 block are required, since there is only three possible PU partitions for a 8×8 luma block, i.e., 8×8, 8×4 and 4×8 (see Section II-A).

As presented in Fig. 3 (*Framework*), the active warp starts by fetching the corresponding sub-block *Motion Data* from the global memory. Then, provided that the block under processing is not encoded with Intra prediction (*Motion Data* bit 5), the *Parallel Interpolation* is performed on a reference

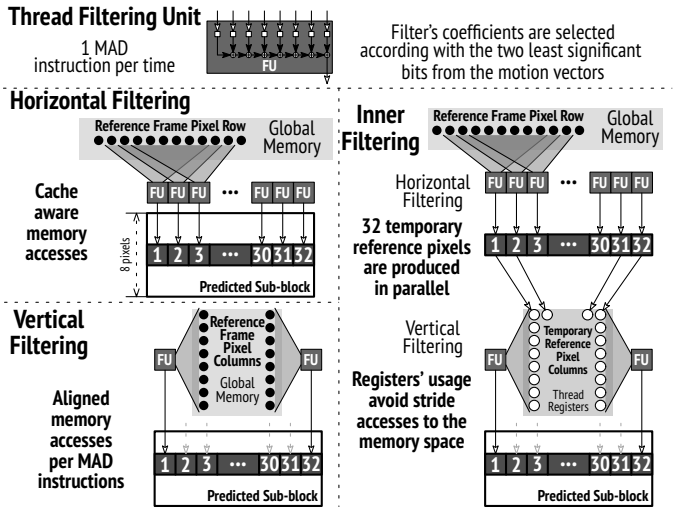


Fig. 4. Filtering unit per thread and proposed parallel interpolation process.

frame from List 0, if L0 is used as reference (bit 6). After the *Parallel Interpolation* on L0, the predicted $N \times 8$ sub-block is kept in the GPU shared memory (*Store Predicted Block*). Since the warps are independent from each other, the GPU shared memory space is used to reduce the GPU register usage and spilling. Afterwards, the same process is repeated for List 1 by checking if the *Motion Data* bit 7 is set. When both picture buffers are selected, the final predicted block is obtained after the *Weight Prediction*, where the average of both sub-blocks is calculated according to the *Weight Factors* stored in the GPU constant memory. To avoid the stridden memory accesses and improve the performance, the whole procedure is repeated until the 64×8 set of sub-blocks is fulfilled in the shared memory, which is subsequently transferred to the global memory.

The *Filtering Unit* (FU) in Fig. 4 illustrates the filtering procedure that is performed by each thread. Herein, one multiply-add (MAD) instruction is executed at each step and the filter coefficients are stored in the GPU constant memory. Each FU requires eight pixels from the reference frame as input to predict one pixel of the sub-block (for 7-tap filtering, one of the filter coefficients is set to zero). The *Horizontal Filtering* in Fig. 4 presents the FU operations performed by each thread in a warp. As it can be observed, for each thread, the input pixel window is shifted by one pixel (at each MAD instruction), which allows efficient use of the GPU cache. For the *Vertical Filtering*, all threads in a warp process in parallel one pixel row at the time, which improves the kernel

performance by allowing row-wise aligned accesses to the GPU global memory, i.e., the column-wise stridden accesses are eliminated. In the case of *Inner Filtering*, the *Horizontal Filtering* is performed first, but the predicted pixels are stored in GPU registers and used as input for the *Vertical Filtering*.

IV. Experimental Evaluation

To experimentally evaluate the efficiency of the proposed GPU algorithm for the IPD module, the set of JCT-VC test conditions were adopted, by using the main profile in *Random Access* (RA) and *Low Delay B* (LD) configurations [18]. Video bitstreams from the highest frame resolution classes A and B were considered, owing to their computational demand. To further challenge the proposed algorithms, an additional set of Ultra HD 4K sequences [19] was also evaluated (class S).

The proposed approach was implemented with CUDA [20] and integrated within the reference HM 15.0 HEVC decoder [21]. In accordance, only the IPD module is handled by the proposed GPU algorithm, while all the remaining HEVC decoding modules are executed on the CPU, with the original HM. For the GPU execution, CUDA Streams [20] are used to overlap the kernel execution and data transfers, where each CUDA stream is responsible for a set of CTU rows.

The efficiency of the proposed GPU parallelization was evaluated in a state-of-the-art NVIDIA GPU with CUDA 7.0, i.e., GeForce GTX 980 @ 1126 MHz (G980). The HM 15.0 decoder was chosen for the baseline comparison, since it is the most commonly used implementation in the literature. In particular, its execution time was obtained on a single core of the Intel® Core™ i7-5960X @ 3.0GHz (referred as CPU). To the best of the authors' knowledge, there are no other state-of-the-art approaches of the HEVC IPD on GPUs that can be used for a direct comparison. Moreover, a direct comparison with the CPU implementation of Chi et al. [7] can not be performed, since their presented results reflect the whole decoder.

Table I presents the experimentally obtained average frame processing time for the HEVC IPD module for each considered test sequence. The presented results include both the kernel execution time and the time to transfer the required data to/from the GPU. Since this evaluation focuses on the efficiency of the IPD algorithms, the processing time corresponding to any other HEVC module, such as the Intra prediction or reconstruction, was not included. In fact, to provide a fair experimental evaluation, all decoded Inter frames with more than 15% of intra predicted blocks were not considered.

The average processing times regarding all recommended Quantization Parameters (QPs) [18] are presented for the *CrowdRun* sequence in both configurations (RA and LD). As expected, the overall processing time decreases with the increase of the QP for both the CPU and the G980. For larger QPs, the encoder tends to choose larger PUs in order to achieve bitrate savings, which results in better cache usage of both architectures. Therefore, only the results for the most demanding QP, 22, are shown in Table I for all the other tested sequences.

As it can be observed in Table I, the proposed GPU-based IPD approach significantly outperforms the CPU-based implementation for all sequences, resolutions, QPs and setups. As expected, class *B* achieves the lowest execution time in both architectures, since it has less PUs to process. The maximum speedup (7.98×) was obtained for the *ParkJoy* sequence in RA configuration, where the proposed algorithm achieves a processing time of 17.60 ms, while the CPU counterpart performs at 140.39 ms. In the LD setup, the highest acceleration (7.51×) was attained for the *BQTerrance* sequence, where average processing times of 39.21 ms and 5.22 ms were obtained with the original HM and the proposed approach, respectively.

In what concerns the real-time capabilities, the proposed algorithm achieves an average frame rate of 56, 128 and 246 fps for classes *S*, *A* and *B*, respectively, with the RA setup and 22 QP. In the LD and same QP, the proposed approach deliveries an average frame rate of 50, 116 and 214 fps for the resolutions 1080p, 1600p and 2160p, respectively, i.e., it allows achieving the real-time processing in all setups.

TABLE I
The HEVC IPD module average frame processing time (in ms).

Class	Sequence	QP	Random Access		Low Delay B	
			CPU	G980	CPU	G980
S 3840×2160	CrowdRun	22	139.57	17.69	140.77	19.49
		27	115.65	16.10	116.37	17.96
		32	103.66	15.32	98.77	16.80
		37	95.20	14.70	85.67	16.14
	InToTree	22	133.69	17.75	137.20	19.74
	ParkJoy	22	140.39	17.60	152.51	20.39
A 2560×1600	Traffic	22	53.07	7.31	55.14	8.37
	PeopleOnStreet	22	60.32	8.26	60.70	8.89
	Nebuta	22	55.61	8.64	57.04	9.01
	SteamLocomotive	22	44.32	7.06	50.08	8.13
B 1920×1080	Kimono	22	27.97	4.00	28.78	4.72
	ParkScene	22	31.72	4.14	34.64	4.95
	Cactus	22	21.98	3.47	22.63	3.98
	BQTerrace	22	34.65	4.67	39.21	5.22
	BasketballDrive	22	27.44	4.23	28.54	4.65

V. Conclusion

An efficient parallel approach of a fully compliant HEVC IPD module was proposed, which exploits the capabilities and resources of modern GPUs by leveraging the fine grain parallel processing opportunities of this time consuming module. To attain the offered performance, all the data accesses were carefully managed in order to exploit the GPU memory hierarchy.

The efficiency of the proposed algorithm was assessed on a state-of-the-art GPU device for an extensive set of computationally demanding frame resolutions (1080p, 1600p and 2160p). The obtained experimental results show that the real-time processing was achieved for all tested sequences and for the most demanding QP, providing an average processing time less than 20.4 ms for Ultra HD 4K video sequences.

Acknowledgment

This work was supported by national funds through FCT (Fundação para a Ciência e a Tecnologia), 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] J. Ohm, G. J. Sullivan, H. Schwarz, T. K. Tan, and T. Wiegand, "Comparison of the coding efficiency of video coding standards – including high efficiency video coding (HEVC)," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1669–1684, Dec. 2012.

[2] F. Bossen, B. Bross, K. Suhring, and D. Flynn, "HEVC complexity and implementation analysis," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1685–1696, Dec. 2012.

[3] G. J. Sullivan, J. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[4] G. Cebrián-Márquez, J. L. Hernández-Losada, J. L. Martínez, P. Cuenca, M. Tang, and J. Wen, "Accelerating HEVC using heterogeneous platforms," *The Journal of Supercomputing*, vol. 71, no. 2, pp. 613–628, 2015.

[5] S. Radicke, J.-U. Hahn, Q. Wang, and C. Grecos, "Bi-predictive motion estimation for HEVC on a graphics processing unit (GPU)," *Consumer Electronics, IEEE Transactions on*, vol. 60, no. 4, pp. 728–736, Nov. 2014.

[6] A. Ilic, S. Momcilovic, N. Roma, and L. Sousa, "Adaptive scheduling framework for real-time video encoding on heterogeneous systems," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

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

[8] C. C. Chi, M. Alvarez-Mesa, B. Juurlink, G. Clare, F. Henry, S. Pateux, and T. Schierl, "Parallel scalability and efficiency of HEVC parallelization approaches," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1827–1838, Dec. 2012.

[9] B. Wang, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink, "Parallel H.264/AVC motion compensation for GPUs using OpenCL," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 25, no. 3, pp. 525–531, Mar. 2015.

[10] D. F. de Souza, N. Roma, and L. Sousa, "Cooperative CPU+GPU deblocking filter parallelization for high performance HEVC video codecs," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, May 2014, pp. 4993–4997.

[11] ——, "OpenCL parallelization of the HEVC de-quantization and inverse transform for heterogeneous platforms," in *Signal Processing Conference (EUSIPCO), 2014 Proceedings of the 22nd European*, Sept. 2014, pp. 755–759.

[12] D. F. de Souza, A. Ilic, N. Roma, and L. Sousa, "Towards GPU HEVC intra decoding: seizing fine-grain parallelism," in *Multimedia and Expo (ICME), 2015 IEEE International Conference on*, July 2015.

[13] ——, in *11th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2015)*, July 2015.

[14] ——, "HEVC in-loop filters GPU parallelization in embedded systems," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XV), 2015 International Conference on*, July 2015.

[15] I.-K. Kim, J. Min, T. Lee, W.-J. Han, and J. Park, "Block partitioning structure in the HEVC standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1697–1706, Dec. 2012.

[16] Y. Yuan, I.-K. Kim, X. Zheng, L. Liu, X. Cao, S. Lee, M.-S. Cheon, T. Lee, Y. He, and J.-H. Park, "Quadtree based nonsquare block structure for inter frame coding in high efficiency video coding," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, no. 12, pp. 1707–1719, Dec. 2012.

[17] K. Ugur, A. Alshin, E. Alshina, F. Bossen, W.-J. Han, J.-H. Park, and J. Lainema, "Motion compensated prediction and interpolation filter design in H.265/HEVC," *Selected Topics in Signal Processing, IEEE Journal of*, vol. 7, no. 6, pp. 946–956, Dec. 2013.

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

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

[20] NVIDIA, *CUDA™ Programming Guide*, NVIDIA, 2015, v7.0.

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