Dynamic Load Balancing for Real-Time Video Encoding on Heterogeneous CPU+GPU Systems

Svetislav Momcilovic, *Member, IEEE*, Aleksandar Ilic, *Student Member, IEEE*, Nuno Roma, *Senior Member, IEEE*, and Leonel Sousa, *Senior Member, IEEE*

***Abstract*—The high computational demands and overall encoding complexity make the processing of high definition video sequences hard to be achieved in real-time. In this manuscript, we target an efficient parallelization and RD performance analysis of H.264/AVC inter-loop modules and their collaborative execution in hybrid multi-core CPU and multi-GPU systems. The proposed dynamic load balancing algorithm allows efficient and concurrent video encoding across several heterogeneous devices by relying on realistic run-time performance modeling and module-device execution affinities when distributing the computations. Due to an online adjustment of load balancing decisions, this approach is also self-adaptable to different execution scenarios. Experimental results show the proposed algorithm's ability to achieve real-time encoding for different resolutions of high-definition sequences in various heterogeneous platforms. Speed-up values of up to 2.6 were obtained when compared to the video inter-loop encoding on a single GPU device, and up to 8.5 when compared to a highly optimized multi-core CPU execution. Moreover, the proposed algorithm also provides an automatic tuning of the encoding parameters, in order to meet strict encoding constraints.**

***Index Terms*—Video Coding, GPGPU, Hybrid CPU+GPU System, Load Balancing.**

I. Introduction

REAL-TIME compression of high quality video is a fundamental prerequisite for modern video services and applications. By efficiently exploiting temporal and spacial redundancy in the video content, the latest coding standards, such as H.264/MPEG-4 AVC [1] and HEVC [2], offer efficient video compression for a wide range of bitrates and resolutions [3]. However, such compression efficiency is offered at the cost of a significant increase of the encoding procedure computational demands, and particularly for certain video coding modules.

Driven by the increasing capabilities of modern compute systems, equipped with general purpose multi-core CPU and GPU, this manuscript is focused on achieving real-time video encoding of HD video sequences by simultaneously processing on all available devices in these hybrid platforms. Several levels of parallelism are considered, which do not only address the parallelization of the video encoding modules on different architectures, but also their efficient collaborative execution in heterogenous environments.

The encoding structure of H.264/AVC and HEVC comprise several modules to process the raw input data. Due to their different characteristics, the efficient parallelization of these modules requires the observance of computational demands and data dependencies at several levels: *i)* between consecutive frames, *ii)* within a single video frame, and *iii)* between the processing modules. Moreover, architecturally different devices in CPU+GPU systems impose additional challenges in this parallelization: while GPUs favor to exploitation of fine-gained data-level parallelism for simultaneous processing on hundreds of cores, CPU architectures, with several general purpose cores, exploit the parallelism even at a coarser-grained level. Therefore, the different modules should be parallelized for each device by considering not only the inherent per-module parallelization potentials, but also the device architectural characteristics. It is also important to properly guide the modules' parallelization to assure an efficient collaborative execution in CPU+GPU systems, by providing the unified per-module video coding functionality across different devices.

In order to fully exploit the synergetic computational potential of CPU+GPU platforms, efficiently parallelized modules have to be additionally integrated into a single cross-device unified execution environment. Although several frameworks, such as OpenCL [4] and StarPU [5], address the programmability issues in CPU+GPU platforms, we focus herein on a direct integration of the parallelized modules by relying on vendor-specific programming models and tools, which also allows a full execution control and attaining a per-device peak performance. Other important issue considered herein deals with efficient scheduling and load balancing in heterogenous CPU+GPU systems. In particular, the attainable performance in multi-core CPUs and GPUs may differ by several orders of magnitude. This difference does not only come from their different architectures, but also from the ability of each device to efficiently process a certain module (device-module affinity). Furthermore, the architecture of current desktop platforms integrates the GPU devices as accelerators, where explicit data transfers must be performed prior and after the GPU execution. The data transfers and the GPU kernel invocations are explicitly initiated by the host CPU and they are conducted over bidirectional interconnection lines (PCI Express) with asymmetric bandwidth. Thus, to fully exploit the capabilities of CPU+GPU systems, it is crucial to consider the performance disparity, module-device affinities and the available bandwidth of communication lines when distributing the loads.

Manuscript received May 15, 2013; revised August 06, 2013; accepted August 06, 2013. Date of publication October 09, 2013; date of current version December 12, 2013. This work was supported by national funds through FCT—Fundação para a Ciencia e a Tecnologia, under projects PEst-OE/EEI/LA0021/2013, PTDC/EEI-ELC/3152/2012 and PTDC/EEA-ELC/117329/2010. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Yen-Kuang Chen.

The authors are with the INESC-ID/IST-TU Lisbon, 1000-029 Lisboa, Portugal (e-mail: Svetislav.Momcilovic@inesc-id.pt; Aleksandar.Ilic@inesc-id.pt; Nuno.Roma@inesc-id.pt; Leonel.Sousa@inesc-id.pt).

Digital Object Identifier 10.1109/TMM.2013.2284892

For the first time, an iterative load balancing approach that allows an efficient employment of all available heterogeneous devices to cooperatively perform the complete video encoding inter-loop procedure [1] is proposed in this manuscript. The proposed scheduling strategy distributes the computations of the parallelized modules across the devices by relying on an online characterization of the computational performance of each device-module pair and the asymmetric bandwidth of the communication lines. These performance parameters are obtained at run-time and without any a priori assumptions of the device, module or communication performance. Furthermore, by iteratively improving the load balancing decisions, the proposed adaptive scheduling method does not only allow achieving a real-time encoding of HD sequences, but also adapting to different execution scenarios. This is particularly important for video coding on unreliable and non-dedicated systems, where the encoding time can greatly vary depending on the current state of the platform (e.g., load fluctuations, multi-user time sharing, operating system actions).

To the best of the authors' knowledge, this is one of the first studies that thoroughly investigates parallel real-time inter-loop video encoding of HD sequences on heterogeneous systems. Its main contributions may be summarized as follows: *(i)* efficient parallelization of all inter-loop modules on both multi-core CPU and GPU devices; *(ii)* integration of the parallelized modules into an unified CPU+GPU execution environment; *(iii)* analysis of the impact of the proposed parallelization algorithms and strategies on the video encoding quality; *(iv)* linear programming-based multi-module dynamic load balancing for multi-core CPU and multi-GPU systems; *(v)* simultaneous iterative scheduling of the entire inter-loop procedure on several heterogeneous devices; *(vi)* accurate online module-specific parametrization of the computational performance and asymmetric communication bandwidth in heterogenous CPU+GPU platforms; *(vii)* self-adaptive approach for non-dedicated and unreliable execution systems.

II. Related Work

Current state-of-the-art approaches for parallel video coding on commodity platforms mainly deal with the individual acceleration of certain modules on multi-core CPU or GPU architectures, such as INT [6], MC [7], (inverse) transform and (de)quantization (T&Q) [8], [9] and DBL [10]. The ME parallelization usually considers FSBM [11]–[18], while only a few studies consider fast algorithms [19]–[21]. In fact, an efficient GPU parallelization of the FSBM is usually achieved by relaxing the spacial data-dependencies, i.e., by redefining the SA center either with zero [12], [13] or with temporary dependent predictors [15], [16] (see Section III). This approach is even used for parallelization of the fast algorithms [20], [21]. In contrast to the FSBM-based approaches, which are capable of fully exploiting the GPU architecture, the implementation of the highly adaptive fast algorithms in GPUs often does not result in significant performance improvements [19], [21]. Only rare attempts on efficient parallelization of the complete encoder (or its main functional parts) have been presented, namely, for multi-core CPUs [22], GPUs [23], or CPU+GPU [24]–[26] environments. In heterogenous CPU+GPU systems, state-of-the-art approaches usually *i)* simply offload one of the inter-loop modules in its entirety (mainly the ME) to the GPU, while performing the rest of the encoder on the CPU [13]–[15], [21], [24], or *ii)* exploit simultaneous CPU+GPU processing at the level of a *single* inter-loop module [11], [25], [26]. However, these approaches have a limited scalability (only one GPU can be employed) and cannot efficiently exploit the full CPU and GPU capabilities (since CPU is idle, while GPU processes the entire offloaded module) [13]–[15], [24]. Furthermore, for the simultaneous CPU+GPU processing of a single-module, the existing methods for cross-device load distribution usually perform exhaustive search over the set of all possible distributions and/or rely on simplified models for module/platform performance. In particular, in [21] the partitioning for "sub-frame" pipelining is decided through a large set of experiments; [25], [26] use a single GPU and constant compute-only performance parametrization; whereas the load distribution in [11] is found by intersecting the experimentally obtained fitted full performance curves. In [17], [18] a simple equidistant data partitioning is applied for video encoding in multi-GPU systems, since the CPU is only used for controlling a homogenous set of GPUs.

The methods proposed herein span over three load balancing/scheduling classes for heterogenous environments, namely: simultaneous multi-module load balancing, static DAG-based scheduling and dynamic iterative load balancing. Since the considered row-based frame partitioning require a distribution of independent computations, the proposed methods are related to multi-application divisible load (DLT) scheduling [27], which usually relies on linear programming to determine the cross-device load distributions [28]. However, there are only a limited number of studies targeting the DLT scheduling in CPU+GPU systems either for general [29], [30] or application-specific [31] problems. The proposed dynamic iterative load balancing routine with on-the-fly update of performance parameters relies on our previous contributions in this area [29], [30]. However, a direct application of functional performance modeling (FPM) [29], [30], [32] was not possible at simultaneous multi-application processing level, since up to date there are no known algorithms to solve this problems with FPM.

The importance of efficient scheduling in video encoding has been already emphasized in [33], where the authors apply DLT scheduling for a single-module load distribution in single-port CPU-only distributed environments for a non-H.264/AVC encoder. However, to the best of the authors' knowledge, the work proposed herein is one of the first that thoroughly investigates dynamic load balancing, scheduling and parallelization of H.264/AVC inter-loop modules for real-time video encoding in heterogenous multi-core CPU and multi-GPU environments.

III. Parallelization of the Video Encoder

An efficient parallelization of the video encoder on heterogenous devices requires a detailed analysis of the entire encoding structure and the parallelization potential of each individual module, regarding their inherent data dependencies and computational demands. The parallelization of the inter-loop modules proposed herein focuses on the H.264/AVC standard, even though the presented strategies can also be applied to the HEVC standard, due to the similar encoding structure,

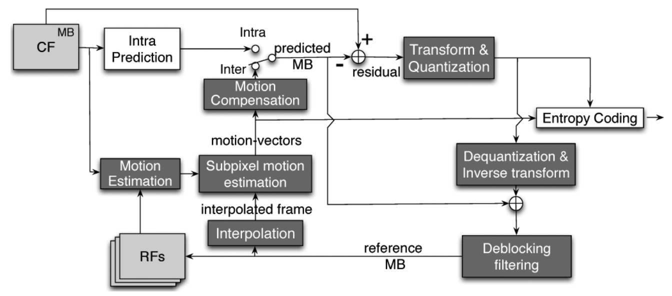


Fig. 1. Block diagram of the H.264/AVC encoder: inter-loop.

modules' functionalities and data dependencies [1], [2]. Furthermore, when compared to the H.264/AVC standard [1], even higher processing speedups are expected from the application of the proposed parallelization models to the several HEVC modules, due to their even higher computational requirements and parallelization potentials.

According to the H.264/AVC standard [1], the CF is divided in multiple square-shaped MB, which are encoded using either an intra- or an inter-prediction mode (see Fig. 1). The standard allows a further subdivision of the MB by considering 7 different partitioning modes, namely 16×16, 16×8, 8×16, 8×8, 8×4 and 4×4 pixels. In the most computationally demanding and frequently applied inter-prediction mode, the prediction of each MB is obtained by searching within already encoded RF. This procedure, denoted as ME, is then further refined with previously interpolated RF from INT module by applying the SME procedure. In the MC module, the residual signal is computed according to the selected MB subdivision mode, which is found as the best trade-off between the data size required to encode the residual signal and the MV. This residual is subsequently transformed and quantized, entropy coded (alongside with the MV and the mode decision data), and finally sent the decoder. The decoding process, composed of the dequantization, inverse integer transform $((\mathrm{T\&Q})^{-1})$ and DBL, is also implemented in the feedback loop of the encoder, in order to locally reconstruct the RF.

A. Data-Dependencies and Parallelization of the Encoder Inter-Prediction Loop

In the H.264/AVC inter-prediction loop, the encoding of the CF can not start before the previous frames are encoded and the required RF are reconstructed, which prevents the encoding of several frames in parallel. Moreover, the inherent data dependencies between the neighboring MB in certain inter-loop modules (see Section III-C) also limit the possibility to concurrently perform the entire encoding procedure on different parts of a frame. Hence, efficient pipelined schemes with several modules can hardly be adopted, either for parts of the frame or for the entire frame. Furthermore, the output data of one module is often the input data for another (e.g., the MVs from ME define the initial search point for the SME), which imposes additional data dependencies between the inter-loop modules. Consequently, the data-dependent inter-loop modules generally have to be sequentially processed (within a single frame). The only exceptions are ME and INT modules, which can be simultaneously processed, since both of them use the CF and/or the RF.

B. Analysis of Efficient Parallel Motion Estimation on CPU + GPU Platforms

Considering its high computational complexity, efficient ME parallelization is crucial to minimize the overall encoding time. Along its execution, a set of candidates within a predefined SA and multiple RF are separately analyzed for 7 different MB-partitioning modes, by using the computationally demanding SAD to calculate the matching distortion. In this procedure, the ME algorithm relies on a median predictor, not only to determine the SA center, but also to compute the displacement of the MV, whose cost is added to the obtained SAD value when computing the matching distortion [1]. In general, there are two classes of ME algorithms, namely FSBM algorithm and fast search algorithms. By relying on exhaustive **FSBM** algorithm, all possible candidates within a predefined SA and multiple RF are separately analyzed. Although it allows achieving very high RD performance, FSBM is often considered impractical when serially performed on a single CPU core due to its high computational complexity. However, the regular processing pattern of FSBM algorithm makes it particularly suitable for fine-grained GPU parallelization [11]–[18], [34], where a significant processing time reduction can be achieved by exploiting the high computational capabilities of GPUs.

In contrast to the FSBM, **fast search algorithms**, such as the EPZS [35] and the UMHexagonS [36], are usually less computationally demanding. Although significantly lower when compared to the FSBM, the execution time when fast algorithms are serially performed on a single CPU core can not meet the real-time requirements, especially for HD sequences and demanding video coding parameters (e.g., several partitioning modes, multiple RFs). However, in contrast to the FSBM, acceleration of the fast ME algorithms on GPU devices is limited by their highly irregular structure and inherent data dependences [19]. In detail, to define both the SA center and the early stopping criterion, these algorithms usually do not only apply the median predictor (as in FSBM), but also a larger set of predictors, including the best MV from neighboring MB. In the case of the EPZS algorithm, the distortion values found for these MV are also used to decide the search pattern and to define the early stopping threshold. This fact seriously limits the efficiency of a parallel multi-thread processing, as current MB determines the ME procedure in the neighboring MB. Since the computational complexity of fast algorithms usually highly depends on the video content, it is hard to balance the distribution of the computational load not only among the different MB, but also across different MV candidates. This is particularly important for GPU parallelization, where it is required to evenly distribute the loads across hundreds of cores. Moreover, the irregular execution pattern in fast algorithms causes frequent branch divergency among the GPU threads [37], leading to the threads serialization and consequently to a degradation of the GPU performance [19], [38].

In order to fully exploit the capabilities offered by different architectures for **collaborative ME on CPU+GPU systems**, it is crucial to ensure that the devices perform the computations on different frame partitions, by relying on different per-device implementations of the ME algorithm. Considering all above-mentioned limitations for GPU parallelization of fast algorithms, which in certain cases might even result in a slowdown when compared to the multi-core CPU execution [19], it was decided to adopt the FSBM algorithm. Furthermore, achieving the load balancing across processing devices in CPU+GPU systems by relying on fast algorithms is almost impossible due to the algorithms' unpredictable performance, which highly depends on the video content. Contrary, since GPUs can usually deliver higher performance than multi-core CPUs, by collaboratively performing the FSBM it is expected to obtain higher acceleration in CPU+GPU systems (see Section V).

Although the FSBM algorithm imposes significantly less data dependencies than the fast algorithms, its efficient parallelization is still limited by the usage of the **median predictor**, which is obtained as the median value of the best matching MV of the left, up, and right-up neighbors. As a consequence, the processing of the current block can not start until these neighbors have been processed. This limits the number of concurrently processed MB to one half of the number of blocks located in the anti-diagonal of the CF [39], which is not sufficient to fully exploit the computational power of the GPUs [37]. Furthermore, besides this restriction imposed by the inter-block dependencies, the use of independent median predictors for each MB partition seriously compromise the efficiency of a GPU parallelization. In particular, a common strategy to decrease the computational demands of the FSBM [11]–[18], [26], [34] is to reuse the SAD values for the smallest MB partitions to hierarchically compute the SADs for larger ones [40]. However, this is only possible when the same predictor is used to determine the SA center for all MB partitions, i.e., they share the same current and reference MBs. Although several works adopt the zero MV as the SA center [12], [13], [26], it is not the only one that can be applied in practice. In detail, the SA center can be determined with any predictor that satisfies the above-mentioned conditions, i.e., it does not impose any additional spatial dependencies. Even though the selection of the predictor does not influence the amount of performed computation in the FSBM (total processing time), it might significantly influence the achieved RD performance. Therefore, the RD performance of a set of possible candidates for SA center predictors was analyzed in [39] according to the Video Coding Experts Group (VCEG) recommendations [41] and by considering a large set of video sequences, resolutions, quantizer values and search ranges. The obtained results suggest that the selection of an adequate predictor is not an easy task, since it highly depends on the sequence characteristics and video coding parameters [39]. However, in order to challenge the real-time inter-loop video encoding in CPU+GPU systems, it was observed that the best MV found for the 16×16 partitioning mode in the previous frame for the collocated MB represents a good compromise for the SA center predictor [39]. It is worth noting that the proposed predictor is only used to compute the SA center. Afterwards, the selected MV are post-computed according to real median vectors of the neighboring MB.

C. Parallelization of Individual Modules

Due to the large number of parallelized modules and limited space in the manuscript, we provide only a brief description of the applied parallelization techniques for GPU and CPU architectures. Further details can be found in [39].

1) Motion Estimation Parallelization: The *CPU parallelization* exploits both coarse-grained and fine-grained data-level parallelism, by processing several MB-rows on different threads and by applying SIMD vector instructions to the processing of neighboring candidates, respectively. The cyclic reuse of reference sample vectors is achieved by processing the candidates in a column major order. When hierarchically computing the matching distortions, a throughput of two $\mathrm{SAD}_{4\times4}$ values per vector instruction is attained. Finally, in order to reduce the branches, the minimum distortion values are determined with a single vector instruction, applied to the concatenated {distortion | MV} pairs. Since the distortion value is placed in the most significant 2-bytes, by updating the minimum distortion, the attached MV is automatically updated. The fine-grained *GPU parallelization* is based on [12], which was not only further optimized, but also additional SA scalability was provided [39]. In detail, the MB are processed on different CUDA thread-blocks, while the matching candidates are examined in parallel on the various threads. The entire MB are kept in the local caches, while the SA samples are cached and processed in portions, according to the available space in the local cache. The distortion values of larger MB-partitions are calculated with hierarchical SAD reusing. The minimum distortion values found among the candidates processed by each individual thread are kept in local registers. Finally, the minimum values found by the different threads are compared with each other in an optimized reduction process, where the memory access pattern and the thread reusing were improved and the thread divergency was minimized [12].

2) Interpolation Parallelization: In the *CPU parallelization*, different MB rows are examined by several parallel threads. Within a single thread, SIMD vector instructions are applied on the sub-pixels with the same offset from the full-pixel, since they require the same filtering operations. However, the complexity of the filtering operation significantly differs between interpolation of half- and quarter-pixels, i.e., six-tap filtering versus linear filtering. In the *GPU parallelization*, each thread-block interpolates the pixels corresponding to a single MB. Within a tread-block, each thread computes all sub-pixels for a single full-pixel [12], [39]. Both the CPU and GPU parallelizations store the interpolated pixels in a specific format to ease the vectorization of the SME module. In the SME module, the same distortion operation is applied on sub-pixels that are exactly 4 sub-pixel positions apart from each other (in the original format of the interpolated frame [1]), where the sub-pixels can be full-, half- or quarter-pixels. However, to increase the efficiency of the proposed parallelization approach, the interpolated frame is subdivided into 4 different *interpolated subframes*, in order to allow the direct application of vector instructions. In particular, since the vector instructions can be applied on data stored in successive memory positions, each *interpolated subframe* consists only of sub-pixels that require the same distortion operation (every fourth sub-pixel from the original format), which are now stored in consecutive memory addresses. With such approach, the vectorized SME distortion calculation can be efficiently applied on sub-pixels belonging to each *interpolated subframe*.

3) Sub-Pixel Motion Estimation Parallelization: In contrast to the full-pixel ME, the SME candidates for different MB-partitions are examined on disjoint SAs centered in the corresponding MVs found in the full-pixel ME procedure. Hence, its parallelization in both CPU and GPU architectures can not exploit neither the hierarchical distortion computation nor the caching of the SA samples (since they can not be reused for multiple MB partitions). The SME is implemented and parallelized in seven separate sub-modules, each one for different MB partitioning granularity by applying different parallelization strategies [39]. In particular, the *CPU parallelization* is performed by exploiting both coarse-grained parallelism (different MB-rows are examined in separate threads) and fine-grain parallelism (within a single tread the distortion is computed by simultaneously processing several MB partitions with vector instructions). The number of simultaneously processed MB partitions is determined by the ratio between the vector register size and the MB partition width, while the total number of vector instructions corresponds to the MB partition height. The overall parallelization efficiency is assured by using seven data-independent kernels in the *GPU parallelization* (one for each SME sub-module), which are simultaneously processed in separate CUDA streams. Each sub-module is parallelized such that the MB are processed by parallel thread-blocks, and each thread processes a single candidate of a single MB-partition. Moreover, only MB samples are cached in the local memory, while the SA samples are accessed directly from the main memory.

4) Motion Compensation Parallelization: The MC module comprises two sub-modules, namely, the MB partitioning mode decision and computation of the residue. In the *CPU parallelization*, both steps process different MB-rows, by exploiting the coarse-grained parallelization. The fine-grained parallelization of the mode decision sub-module uses vector additions to accumulate the distortion values for different partitioning modes, which are then compared to find the mode with the minimum distortion. According to the selected mode, the residual signal is computed by applying a vectorized subtraction to the selected best matching candidates from the current MB partitions. In the *GPU parallelization* of both sub-modules, a single thread block examines one MB. In the mode decision kernel, several reduction trees are applied to accumulate the distortions for each mode and to determine the minimum distortion mode. The residual signal is computed within a single thread, by using the loaded pair of the original and reference pixels [39].

5) Inverse Transform and de Quantization Parallelization: The coarse-grained *CPU parallelization* assigns different MB-rows to different threads for T&Q and $(\mathrm{T\&Q})^{-1}$ modules. Between the vertical and horizontal transforms, the transposition of the considered MB partitions is performed, where both transform and transpose operate on two MB partitions at once. The (de)quantization process is performed by applying a vectorized multiplication and shift operations on transformed/quantized samples in subsequent memory locations [39]. In the *GPU parallelization*, a single MB is processed in a single thread block. When the transform is performed along the vertical direction, each thread computes a single column, while in the horizontal direction it computes a single row. To reuse computed pixel offsets, quantization and its inverse are performed within the same kernel.

6) Deblocking Filtering Parallelization: The DBL is performed in two steps to filter vertical and horizontal edges. Due to the strict data dependencies between the MB sharing the same edge, the wavefront model [39], [42] is usually adopted. Accordingly, the coarse grained *CPU parallelization* is applied to different MB on a single anti-diagonal. On the other hand, the DBL vectorization is limited by the highly adaptive nature of the algorithm, where different filtering operations need to be applied for different edge pixel values. Consequently, the vectorization can only be done if all possible branches are executed and the correct ones are selected afterwards, according to the observed branch conditions. In order to vectorize the filtering of vertical edges, it is also required to transpose the loaded vectors. The application of the wavefront pattern to a fine-grained *GPU parallelization* seriously limits the attainable GPU performance, since the number of MB in the anti-diagonal (thread blocks) is not sufficient to fully exploit the GPU computational power. In the initial step, each thread filters one pixel of the vertical edge, which is subsequently used for the processing of the horizontal edges. Despite all considered optimizations [39], this algorithm cannot avoid branch divergence, but it takes advantage of adaptivity to reduce the computation load.

IV. Collaborative Video Encoding On Multi-Core CPU and Multi-GPU Environments

As depicted in Fig. 2, modern CPU+GPU systems incorporate a set of k CPU cores and w GPU accelerators, i.e., p_i heterogenous devices, where $i=\{1,..,k+w\}$. Since the accelerators are not stand-alone devices and usually perform the computations on data explicitly transferred from the main memory, the CPU is responsible for initiating both the device executions and the data-transfers across the interconnection lines. Hence, in order to fully exploit the capabilities of CPU+GPU systems for collaborative video encoding, an unified execution environment is developed, which integrates different per-device parallelizations of the inter-loop modules presented in Section III. This environment ensures the efficient cross-device execution by coalescing different programming models and vendor-specific techniques. For such purpose, it provides the full execution control in terms of per-device memory management, automatic mechanisms for data-transfers between the devices, as well as the necessary kernel launches. Since a module might fit better to a certain device architecture (GPU or CPU), one of the clear benefits of such an unified environment is that the processing of the inter-loop modules can be distributed to different devices with distinct characteristics, according to their module-device affinities. However, an additional level of collaborative execution is also considered herein, since several heterogenous devices can simultaneously perform different MB rows of a single module. Hence, we tackle herein the issues related to the efficient cross-device parallel inter-loop processing at three levels, namely: *i)* simultaneous collaborative processing; *ii)* inter-module scheduling and *iii)* dynamic iterative load balancing.

Simultaneous collaborative processing can be efficiently performed on the inter-loop modules with sufficient amount

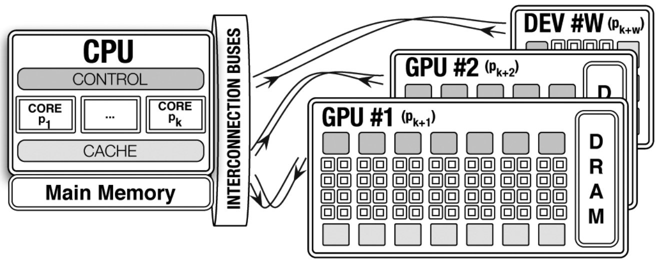


Fig. 2. Heterogeneous multi-core CPU and multi-GPU/accelerator system.

of computational load to distribute. According to the experimental evaluation provided in [39], ME, INT, and SME modules take more than 90% of the total inter-loop encoding time, which makes them good candidates for this kind of processing. Hence, it is necessary to determine the most suitable distribution (e.g., the number of MB rows) across the heterogeneous devices, such that the overall execution is as balanced as possible. In detail, for both the ME and INT modules, the distribution vector $\alpha=\{(m_i,l_i)\}$ needs to be determined, where the tuple (m_i,l_i) represents the number of MB rows assigned to ME (m_i) and INT (l_i), respectively. Similarly, for each SME kernel $(j\in\{16\times16,16\times8,8\times16,8\times8,8\times4,4\times8,4\times4\})$, it is required to determine the distribution vector $\beta=\{(s_{i,j})\}$ with the number of MB rows $(s_{i,j})$ to be processed on each p_i device. Due to the lower impact of the remaining modules on the total video encoding time, the adopted inter-module scheduling assigns each remaining module to a device according to its module-device affinities, such that the overall and sequentially performed $\mathrm{MC}+\mathrm{T\&Q}+\mathrm{T\&Q}^{-1}+\mathrm{DBL}$ procedure is executed in the shortest time.

The proposed **iterative approach for scheduling and load balancing** does not rely on any assumption about module or device performance. Instead, it relies on realistic performance characterization which is obtained during the execution of the video coding procedure (in runtime). In order to capture the performance disparity of the heterogeneous devices and the corresponding module-device affinities, the performance of each device is quantitatively expressed as the ratio between the given workload (e.g., MB rows) and the time taken to process it. In detail, for the ME, INT and SME modules, the performance of each p_i device is expressed by μ_i, λ_i and $\sigma_{i,j}$ for each module/mode, respectively (according to the previously defined m_i, l_i, and $s_{i,j}$ distributions). Since the bandwidth of the interconnection lines does not depend on the module characteristics, it is expressed as the ratio between the data size (i.e., $bytes$) and the time taken to transfer the data between the CPU and the connected device. Due to the fact that modern bidirectional interconnections (e.g., PCI Express) usually deliver an asymmetric bandwidth, the available bandwidth from the CPU to the device is modeled with ι_i, while θ_i expresses the bandwidth from the device to the CPU, for each connected device $p_i(i=\{k+1,..,k+w\})$. The general structure of the proposed algorithm for collaborative inter-loop video encoding consists of two main routines:*Initialization phase* and *Iterative phase* (see Algorithm 1).

Algorithm 1 Inter-Frame Scheduling Algorithm (Outline)

```
Initialization phase
1: load the first inter-frame
2: determine initial $\alpha$ distribution for ME and INT, such that $m_i=l_i=N/(k+w)$ for each processor $p_i$, $i=\{1,..,k+w\}$
3: execute, in parallel, the assigned ME and INT MB rows ($m_i$ and $l_i$) on each $p_i$ device and record the execution time of each load ($tm_i$ and $tl_i$), as well as the time taken to transfer all the input and output data
4: determine initial $\beta$ distribution by equidistant load partitioning $s_{i,j}=N/(k+w)$ for each processor $p_i$ and for each SME mode $j\in\{16\times16,16\times8,8\times16,8\times8,8\times4,4\times8,4\times4\}$
5: execute the assigned $s_{i,j}$ loads and record the corresponding execution times $ts_{i,j}$ and the transfer time for input and output data, for each device and SME mode
6: on each $p_i$ device, perform the remaining inter-loop modules and record the execution times, as well as the input and output data transfer times
7: calculate the per-processor performance $\mu_i=m_i/tm_i$, $\lambda_i=l_i/tl_i$, and $\sigma_{i,j}=s_{i,j}/ts_{i,j}$, as well as the asymmetric bandwidth of the interconnections, $\iota_i$ and $\theta_i$, for each non-CPU device $p_i$ ($i=\{1,..,k+w\}$), according to the recorded transfer sizes and times in each direction
8: proceed to the Iterative phase
Iterative phase:
1: for $frame\_nr=2$ to $nr\_of\_inter\_frames$ do
    2: call ME+INT Load Balancing routine to determine the $\alpha$ distribution
    3: simultaneously process the assigned number of MB rows for ME and INT modules (from $\alpha$ distribution) on each $p_i$ device, perform the input and output transfers, and record the corresponding times
    4: call SME Load Balancing routine to determine the $\beta$ distribution
    5: simultaneously execute SME modes (from $\beta$ distribution), perform the necessary transfers, and record the execution and transfer times
    6: call Inter-module scheduling routine to determine the module-device execution pairs for the remaining modules and record the corresponding processing/transfer times
    7: Update Relative Performance Parameters and determine if the load balancing is achieved (for dedicated systems)
8: end for
```

A. Initialization Phase

The *Initialization phase* of the algorithm is used to assess the initial performance parameters, obtained during the encoding of the first inter-frame. For the *simultaneous collaborative processing* of ME, INT and SME modules, the required performance information is obtained by performing an equidistant MB row-wise partitioning among all available heterogeneous devices (lines 2 and 4 in Algorithm 1). The execution times on each device, as well as the input and output data-transfer times, are recorded (lines 3 and 5) and used to calculate the module-specific relative device performance and asymmetric bandwidth of interconnection lines (line 7). For *inter-module scheduling*, the remaining modules are run, in parallel, on all the existing heterogeneous devices, in order to assess the module-device affinities, by recording the execution time for each module (line 6). The obtained data in the *Initialization phase* is used as an input of the *Iterative phase*.

B. Iterative Phase

After the *Initialization phase* , which is only applied during the encoding of the first inter-frame, the *Iterative phase* is applied for every subsequent inter-frame. This phase dynamically balances the computational load of the simultaneously processed modules, and efficiently distributes the remaining modules. Moreover, it also iteratively improves the load balancing decisions and adapts to the current state of the execution platform. The *Iterative phase* consists of three main scheduling routines: *1) ME+INT Load Balancing* (line 2 in Algorithm 1); *2) SME Load Balancing* (line 4), and *3) Inter-module scheduling* (line 6), which are all used for each inter-frame. The scheduling decisions from these three routines are used for parallel cross-device execution of the respective modules, followed by recording the execution/transfer times for each assigned module-device pair (lines 3, 5 and 6). The *Update Relative Performance Parameters* (line 7) concludes the *Iterative phase* for a single inter-frame, by updating the performance parameters and determining the achieved balance.

1) $ME + INT$ Load Balancing: The simultaneous collaborative processing of ME and INT modules relies on linear programming to obtain the $\alpha = \{(m_i, l_i)\}$ distribution, by partitioning the total number of MB rows (N) across the available heterogenous devices. Since each (m_i, l_i) distribution tuple consists of strictly integer values, the load balancing approach relies on relaxing an integer linear program into a rational formulation [28], in order to reduce the scheduling overheads while ensuring a parallel execution as balanced as possible. This relaxation allows to obtain the *upper bound* of the optimal load distribution, which is transposed into the discrete domain by applying a refinement procedure.

The load balancing problem considered herein is usually denoted as *multi-application divisible load scheduling*, whose linear program formulation is summarized as follows: MINIMIZE $T_{\mathbb{MI}}$ SUBJECT TO:

$$m_i \geq 0;\, l_i \geq 0;\, \sum m_i = N;\, \sum l_i = N \quad (\forall i \in \{1,..,k+w\}) \tag{1}$$

$$\frac{m_i}{\mu_i} + \frac{l_i}{\lambda_i} \leq T_{\mathbb{MI}} \quad (\forall i \in \{1,..,k\}) \tag{2}$$

$$\frac{m_i \cdot b_{cf}}{\iota_i} + \frac{m_i}{\mu_i} + \frac{m_i \cdot b_{mv}}{\theta_i} \leq T_{\mathbb{MI}} \ (\forall i \in \{k+1,..,k+w\}) \tag{3}$$

$$\frac{l_i}{\lambda_i} + \frac{l_i \cdot b_{sf}}{\theta_i} \leq T_{\mathbb{MI}} \quad (\forall i \in \{k+1,..,k+w\}). \tag{4}$$

The objective of this function is to minimize the total ME+INT processing time ($T_{\mathbb{MI}}$) across all employed devices. Equation (1) states that each load within a (m_i, l_i) tuple in the α distribution vector must be of a non-negative value, and that the sum of per-module assigned loads must be equal to N. Equations (2)–(4) reflect the inter-device load balancing conditions (in the time domain), where the parallel execution across several devices (including the data-transfers) should guarantee the minimum $T_{\mathbb{MI}}$. For each module, the computation time is represented by the ratio between the assigned load (m_i or l_i) and the corresponding module-specific device performance (μ_i or λ_i). Although ME+INT are simultaneously performed in all the devices, on the CPU side they are processed in sequence (see (2)), while in each GPU device these two modules are performed in parallel (see (3) and (4)). The time to perform the data-transfers is modeled as the ratio between the amount of transferred data (in $bytes$) and the bandwidth of the interconnection lines in each direction (ι_i and θ_i). The ME module operates on m_i MB-rows from the CF, whose data size is represented as a multiple of m_i and the MB row size b_{cf} (in $bytes$). Similarly, the returned amount of data is represented by the size of produced MV for all MB partitions and MB in a row (b_{mv}), multiplied by the m_i rows that are processed. The amount of data corresponding to the interpolated samples is a multiple of l_i and the size of a single interpolated row b_{sf}. Due to the limited memory of GPU devices and in order to further exploit data reusing, the list of required RF is kept updated in form of a FIFO circular buffer, where the oldest RF is updated with the last RF produced at the end of the inter-loop.

The obtained *upper bound* of the optimal load distribution (MB rows assigned to each device) is a vector of real values, which are subsequently rounded down to the nearest integers. This results in a certain number of loads being unassigned, which cannot exceed the number of employed devices. Hence, the remaining loads are assigned to devices by iteratively incrementing their load until all N rows are distributed for each module, with following refinement procedure:

1. **If** $\sum m_i < N$, $i \in \{1,..,k+w\}$ **then** go to step 2 **else** go to step 4.
2. Find $q \in \{1,..,k+w\}$ such that $(m_q+1)/\mu_q = \min\{(m_i+1)/\mu_i\}$.
3. $m_q = m_q + 1$ **Repeat** step 1.
4. **If** $\sum l_i < N$ **then** go to step 5 **else** stop the refinement.
5. Find $r \in \{1,..,k+w\}$ such that $(l_r+1)/\iota_r = \min\{(l_i+1)/\iota_i\}$.
6. $l_r = l_r + 1$ **Repeat** step 4.

This α distribution is then used for ME+INT collaborative processing on all heterogenous devices.

2) SME Load Balancing: This routine aims at determining the $\beta = \{(s_{i,j})\}$ distribution vector across all p_i devices and for each SME mode $\mathbb{S}_j$, where $j \in \{16\times16, 16\times8, 8\times16, 8\times8, 8\times4, 4\times8, 4\times4\}$. Similarly to the ME+INT load balancing, the β vector is determined by

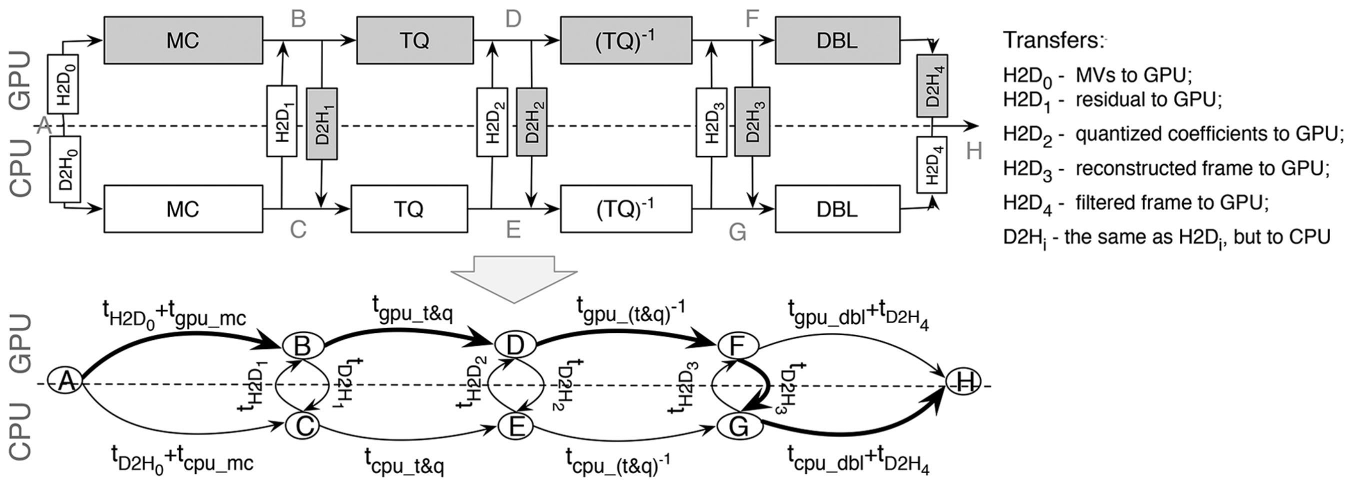


Fig. 3. Construction of a weighted DAG from the data-flow diagram and a possible minimal path (represented in bold).

relaxing an integer linear program into the rational domain and by applying the refinement procedure. The SME load balancing is expressed in the following linear program: MINIMIZE $T_{\mathbb{S}}$ SUBJECT TO:

$$s_{i,j}{\ge}0;\ \sum s_{i,j}{=}N \quad (\forall i{\in}\{1,..,k{+}w\},\forall j) \tag{5}$$

$$\sum_j \frac{s_{i,j}}{\sigma_{i,j}} \le T_{\mathbb{S}} \quad (\forall i{\in}\{1,..,k\},\forall j) \tag{6}$$

$$\frac{b'_{mv}{+}b'_{sf}}{\iota_i}+\frac{s_{i,j}}{\sigma_{i,j}}+\frac{s_{i,j}{\cdot}b_{sv_j}}{\theta_{i,j}} \le T_{\mathbb{S}} \quad (\forall j, i{\in}\{k{+}1,..,k{+}w\}). \tag{7}$$

As before, the main objective is to minimize the total execution time $T_{\mathbb{S}}$ across all employed devices, where (5) states that the loads in each $(s_{i,j})$ tuple are non-negative and their sum is equal to N, for each SME mode. Equations (6) and (7) express the load balancing conditions to attain the minimum $T_{\mathbb{S}}$. For each mode, the computation time is expressed as the ratio between the $s_{i,j}$ load and the corresponding $\sigma_{i,j}$ device performance. The entire SME is processed in parallel on all devices. On the CPU side all $\mathbb{S}_j$ SME modes are sequentially performed (see (6)), while on each GPU device these modes are performed concurrently (see (7)). Prior to the execution of the SME modes on non-CPU devices, the already estimated full-pixel MV and the interpolated samples need to be transferred (i.e., b'_{mv} and b'_{sf}, respectively). For each mode, the resulting amount of data to be transferred is represented by multiplying $s_{i,j}$ by the size of the estimated MV in a single row b_{sv_j} (in $bytes$). To round down the obtained values to the nearest integers, the refinement procedure is then applied to the obtained β load distribution, which is defined as follows:

```
1. For each $j$ from $\mathbb{S}_j$
2. If $\sum s_{i,j}<N$, $i\in\{1,..,k+w\}$ then go to step 3 else go to step 1
3. Find $q\in\{1,..,k+w\}$ such that $(s_{q,j}+1)/\sigma_{q,j}=\min\{(s_{i,j}+1)/\sigma_{i,j}\}$
4. $s_{q,j}=s_{q,j}+1$. Repeat step 2.
```

3) Inter-Module Scheduling: In this procedure, each of the least computationally intensive modules (MC, T&Q, $(\mathrm{T\&Q})^{-1}$, DBL) is mapped into a processing device according to its module-device affinities. To minimize the time of a $\mathrm{MC}+\mathrm{T\&Q}+\mathrm{T\&Q}^{-1}+\mathrm{DBL}$ sequence, this procedure relies on the performance parameters from the *Initialization phase*.

The implementation of this procedure is illustrated in Fig. 3 for the case of a typical CPU+GPU system. To reflect the different module-device execution affinities, a data-flow diagram is dynamically constructed, including both the processing and the data transfer times (in each direction) for each of the remaining modules and devices in the system. This diagram allows the construction of a weighted DAG, which encapsulates all possible communication paths between the accelerators and the CPU. The DAG nodes, i.e., $A, B, .., H$, represent the decision points, where each module can be mapped to any one of the processing devices. The edges represent the individual module transitions, weighted by the respective computing and data transfer times. Thus, the shortest path between the starting and ending nodes corresponds to the minimum encoding time of a complete $\mathrm{MC}+\mathrm{T\&Q}+\mathrm{T\&Q}^{-1}+\mathrm{DBL}$ sequence. Dijkstra's algorithm [43] is typically used to find such a shortest path, determining the best mapping between modules and devices. Due to the small number of nodes and edges, this algorithm does not introduce a significant overhead to the procedure.

4) Relative Performance Parameters Update: As soon as each inter-frame is encoded, the module-specific performance parameters of each device are updated. In particular, by relying on the measured execution and data-transfer times for each device, the $\mu_i, \lambda_i, \sigma_{i,j}, \iota_i$, and θ_i are updated according to the assigned loads in the previously used α and β distributions. Then, the newly calculated performance parameters are used to update the α and β load distributions to be applied in the following inter-frame encoding. Such online updating procedure allows to iteratively improve the load balancing decisions with each processed inter-frame and to adapt to the individual non-linear device performance for the different modules and load sizes. For non-dedicated systems, this procedure provides an important self-adaptability characteristic to the proposed algorithm, which makes it suitable even for execution scenarios where the performance greatly varies with time (as presented in Section V).

TABLE I
PROCESSING DEVICES AND CONSIDERED HETEROGENEOUS CPU + GPU SYSTEMS.

Devices	CPU_N	CPU_C	GPU_T	GPU_F	GPU_F$_1$
Model	Intel Core i7	Intel Core 2 Quad	GeForce 285GTX	GeForce 580GTX	GeForce 580GTX
# Cores	4	4	240	512	512
Frequency	3 GHz	2 GHz	1.48 GHz	1.54 GHz	1.59 GHz
Memory	4 GB	4 GB	1 GB	1.5 GB	1.5 GB

Systems	CPU	GPUs	
SysNT	CPU_N	GPU_T	
SysNF	CPU_N	GPU_F	
SysCF$_1$	CPU_C	GPU_F$_1$	
SysNTT	CPU_N	GPU_T	GPU_T
SysNFT	CPU_N	GPU_F	GPU_T
SysNFF	CPU_N	GPU_F	GPU_F

TABLE II
OVERALL RD ANALYSIS RESULTS FOR DIFFERENT SEARCH AREA SIZES (SA) PERFORMED ACCORDING TO [41].

			SA1: 32x32				SA2: 64x64				SA3: 128x128				SA3 vs. SA1			
Format	Sequence	Encod. frames	Proposed		UMHexagonS		Proposed		UMHexagonS		Proposed		UMHexagonS		Proposed		UMHexagonS	
			dB	%	dB	%	dB	%	dB	%	dB	%	dB	%	dB	%	dB	%
CIF (352x288)	Foreman	300	**-0.31**	**7.34**	-0.36	8.49	**-0.29**	**6.86**	-0.31	7.26	**-0.19**	**4.40**	-0.32	7.47	-0.14	3.33	-0.06	1.46
	Mobile	300	**-0.04**	**0.81**	-0.07	1.46	**-0.05**	**0.97**	-0.06	1.25	**-0.03**	**0.50**	-0.07	1.39	-0.01	0.18	0.00	-0.06
	Paris[1]	300	-0.52	9.76	-0.14	2.61	-0.41	7.69	-0.13	2.35	-0.35	6.57	-0.12	2.28	-0.16	2.85	-0.01	0.19
	Stefan	260	**-0.36**	**6.71**	-1.61	33.09	**-0.36**	**6.71**	-0.60	11.68	**-0.30**	**5.69**	-0.47	9.08	-0.16	2.97	-1.24	24.46
	Table	260	-0.28	6.74	-0.24	5.68	**-0.21**	**4.98**	-0.24	5.72	**-0.15**	**3.54**	-0.23	5.41	-0.14	3.29	-0.02	0.45
	Tempete	260	**-0.10**	**1.85**	-0.30	5.83	**-0.05**	**0.98**	-0.21	4.03	**-0.04**	**0.77**	-0.22	4.20	-0.07	1.33	-0.09	1.82
720p (1280x720)	BigShips	150	**-0.03**	**0.93**	-0.09	3.10	**-0.03**	**0.88**	-0.09	3.18	**-0.03**	**0.88**	-0.09	3.18	-0.01	0.36	-0.01	0.24
	City_corr	150	**-0.06**	**1.87**	-0.20	6.11	**-0.05**	**1.37**	-0.19	5.88	**-0.04**	**1.36**	-0.19	6.06	-0.03	0.72	-0.01	0.26
	Crew	150	-0.18	6.41	-0.14	4.89	-0.19	6.61	-0.14	4.93	-0.19	6.58	-0.14	4.93	-0.01	0.44	-0.02	0.58
	CrwdRun	150	**-0.07**	**1.30**	-0.18	3.52	**-0.07**	**1.37**	-0.18	3.40	**-0.09**	**1.76**	-0.17	3.34	0.03	-0.56	0.00	0.08
	Jets[2]	150	**0.62**	**-11.27**	-1.08	25.19	**-0.16**	**3.04**	-1.37	33.21	**-0.40**	**8.29**	-0.88	20.40	-0.22	4.50	-1.28	32.51
	Night	150	**-0.33**	**9.05**	-0.38	10.51	-0.27	7.48	-0.24	6.56	**-0.21**	**5.65**	-0.21	5.86	-0.17	4.48	-0.21	5.67
	Raven	150	**-0.22**	**5.56**	-0.53	14.17	**-0.20**	**5.11**	-0.47	12.53	**-0.21**	**5.21**	-0.43	11.69	-0.02	0.55	-0.10	2.49
1080p (1920x1080)	RollToms	60	**-0.09**	**4.95**	-0.14	8.60	**-0.11**	**5.97**	-0.16	9.60	**-0.11**	**6.22**	-0.17	10.13	-0.04	2.19	-0.03	2.00
	Sunflower	125	**-0.15**	**4.16**	-0.81	25.72	**-0.08**	**2.27**	-0.45	13.71	**-0.08**	**2.20**	-0.44	13.32	-0.09	2.67	-0.39	11.81
	Toys&Cal	125	**-0.18**	**6.68**	-0.86	38.00	**-0.17**	**6.19**	-0.75	32.72	**-0.15**	**5.54**	-0.76	32.92	-0.05	2.12	-0.12	5.09

[1] FrameSkip=1; [2] StartFrame=300

Whenever the performance does not significantly vary (e.g., dedicated computing systems), it is even possible to reduce the already small impact of the scheduling overhead on the whole encoding time, by terminating the *Iterative phase* as soon as the load balancing is achieved. For each scheduling stage (corresponding to the above-mentioned routines), the achieved load balancing is evaluated by comparing the obtained execution times of the employed devices. If the difference between the measured execution times satisfies a predefined accuracy margin, the achieved load distribution is marked as balanced and used for the encoding of the subsequent inter-frames.

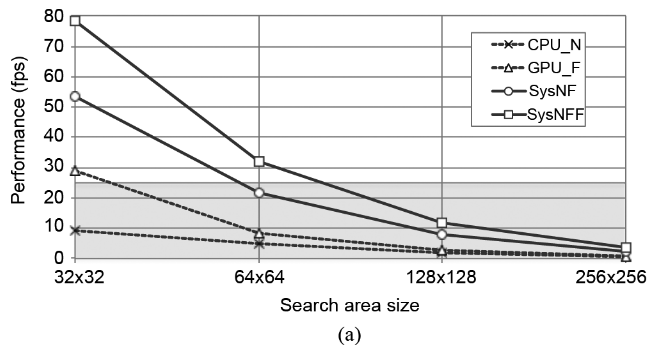
V. EXPERIMENTAL RESULTS AND EVALUATION

To evaluate the proposed parallelization and load balancing/scheduling techniques, this section presents a comprehensive set of experimental results obtained for both the RD and the processing performance of the parallel implementations of the H.264/AVC encoder based on JM 18.4 reference software [44]. For such purpose, a vast set of 352×288 (CIF), 720p, 1080p and even 3840×2160 resolutions were considered. In order to validate the efficiency of the proposed framework and algorithms in several different execution scenarios, a broad set of heterogeneous CPU+GPU setups were specifically chosen to cover a wide range of processing platforms not only with different computational performance, but also with different device architectures (see Table I). In particular, an Intel Core 2 Quad (*CPU_C*) and an Intel Core i7 (*CPU_N*) architectures were used as CPUs, whereas three NVIDIA GPU devices were selected from Tesla (*GPU_T*) and Fermi (*GPU_F*, *GPU_*F$_1$) architectures. All systems were running OpenSUSE 12.1 with CUDA 4.1, Intel Parallel Studio 12.1, and OpenMP 3.0.

A. Rate-Distortion Analysis

As discussed in Section III, an efficient parallelization of the FSBM ME module in GPU-based platforms can be achieved by relying on a single SA center predictor for all MB partitions. Table II shows the overall RD performance obtained for a broad set of video sequences, resolutions, QP and SA sizes. The presented values represent a subset of the extensive RD analysis conducted in [39], where several possible candidates for the SA center predictors were evaluated. These results were obtained by strictly following the VCEG recommendations [41] for $IPPP$ sequences with 4 RF, Baseline Profile, and the following QP {ISlice, PSlice}: {22,23}, {27,28}, {32,33} and {37,38}.

According to [39], the MV that was found in the previous frame for the collocated MB when using the 16×16 partitioning mode (marked with *Proposed* in Table II) can be selected as a viable and advantageous SA center predictor for the current frame MB. The achieved RD performance with *Proposed* predictor was then compared with the performance of the original UMHexagonS [36] algorithm from the JM reference software [44], since one of the rare attempts to parallelize an adaptive fast search algorithm on GPU was made for a simplified version of this algorithm (smpUMHexagonS) [19], which is expected to deliver lower RD performance. To provide a fair comparison between the FSBM ME algorithm that uses the *Proposed* predictor and the UMHexagonS [36] fast algorithm, separate RD-curves were drawn (for different QP) and using the original JM software, under the same testing conditions [39]. The RD analysis was performed by calculating the difference between each of these curves and the RD-curve obtained with the original FSBM algorithm. Table II reports these objective differences calculated by relying on the Bjøndegard measurement method [45] and by using the VCEG software tool [41]. The average differences are expressed in both bitrate (in %) and PSNR (in dB). As it can be observed, for different sequences and SA, the FSBM encoding with the *Proposed* predictor outperforms the UMHexagonS [36] in 40 (out of 48) cases, especially for the HD sequences ($\approx$ 87%). The most notable difference occurs for Jets 720p



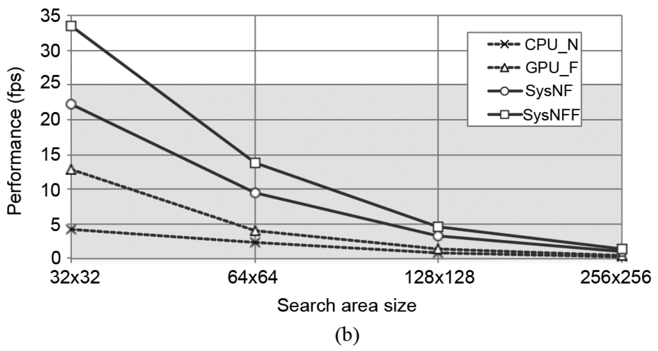


Fig. 4. Achieved performance (in fps) for different SA sizes (average results for 1 RF). (a) 1280 × 720 video resolution. (b) 1920 × 1080 video resolution.

video sequence for small SA, where even the original FSBM was outperformed. This fact can be justified by the presence of a pan-left movement of the camera in the considered part of this low motion video sequence, which allows more accurate motion prediction by relying on the MVs from the previous frames (used by the *Proposed* predictor) than with spatially dependent or zero predictors. However, when larger SAs were considered, even ME with spatially dependent predictors was capable of finding the minimal distortion candidates, which were not within the smaller SA. In order to challenge the real-time encoding on commodity systems (see Table I), the difference between the RD-curves was evaluated for different SA sizes (*"SA3 vs. SA1"*). With the *Proposed* solution, the average difference in bitrate of 1.9% is achieved for all sequences (max. 4.5%) [41].

Finally, although the obtained results clearly show that the proposed predictor achieves an adequate coding efficiency for collaborative CPU+GPU real-time encoding of HD sequences, it is worth to note that further research should be performed in order to determine the best FSBM-based predictor. However, such an investigation is out of the scope of this paper.

B. Evaluation of the Proposed Methods in Multi-Core CPU and Multi-GPU Environments

Fig. 4 presents the obtained performance, measured in encoded fps, of the proposed collaborative inter-loop video encoder. In order to conduct a strict performance evaluation of the video encoder inter-loop, the intra-mode block prediction in all encoded inter frames was disabled and the SME was performed for all MB partitions and partitioning modes. The presented results were obtained in the *SysNF* and the *SysNFF* heterogeneous systems (see Table I) and in a single device setups, i.e., a multi-core CPU (*CPU_N*) and a GPU (*GPU_F*), when processing HD video sequences with different resolutions for four **different SA sizes**. In all presented charts, the shaded area represents the performance region where it is not possible to achieve a real-time encoding. As expected, the overall performance of the inter-loop encoding significantly decreases between two successive SA sizes, due to the quadruplication of the ME computational load. However, it can also be observed that the obtained performance is not four times lower, due to the fact that the complexity of other inter-loop modules does not increase with the SA size. For all the considered SA sizes, the proposed methods succeeded to efficiently exploit the synergetic performance of heterogeneous devices, by significantly outperforming their corresponding single device executions. At this respect, it is also worth noting that, by relying on the proposed collaborative execution strategies, a real-time inter-loop encoding can be achieved even on commodity CPU+GPU desktop systems. In particular, in the *SysNFF* platform, the real-time encoding can be attained for a SA size of up to 64 × 64 pixels for 720p sequences, while it is achieved for 32 × 32 SA size for 1080p sequences. Moreover, these charts also emphasize the efficiency of the proposed GPU inter-loop parallelization, evidenced by the achieved real-time 720p HD encoding on a single *GPU_F* device, with a 32 × 32 SA size. To further challenge the real-time inter-loop encoding on the off-the-shelf desktop systems, we adopt a 32 × 32 pixels SA for the following experimental evaluations, which for the conditions considered herein does not impose a significant RD degradation when compared to larger SA sizes (see Section V-A).

Fig. 5 presents the performance of the proposed methods obtained with different platform configurations (see Table I), when encoding 720p and 1080p HD video sequences for a **different number of RF**. As it can be observed, for these two HD resolutions, the proposed parallelization, load balancing and scheduling approaches allow achieving a real-time encoding rate ($\approx$ 25 fps) for multiple RF. As an example, for the 1280 × 720 resolution, all tested platforms succeeded in achieving a real-time video encoding for multiple RFs. In detail, it can be observed that the proposed methods allow achieving a real-time encoding performance for 12 and 3 RF for the 720p and 1080p resolutions, respectively. As expected, the platforms equipped with Fermi-based GPU devices (*GPU_F*, *GPU*_F_1) tend to deliver a higher overall system performance than the platforms with Tesla GPUs (*GPU_T*). By relying on the presented load distribution strategy, an average speedup of about 2.6 is obtained in the multi-GPU *SysNFF* platform, when compared to the single *GPU_F*, and up to 8.5 when compared to the multi-core *CPU_N* execution, for all considered resolutions and RF.

Although the proposed load balancing and scheduling strategies adapt to the performance variations in the system during the run-time, the scheduling decisions highly depend on the problem granularity to be distributed. In contrast to the computationally intensive modules for which the fine-grained load balancing is performed on a large set of MB rows, the remaining modules are mapped in their entirety to the best performing devices and these mappings usually do not significantly change

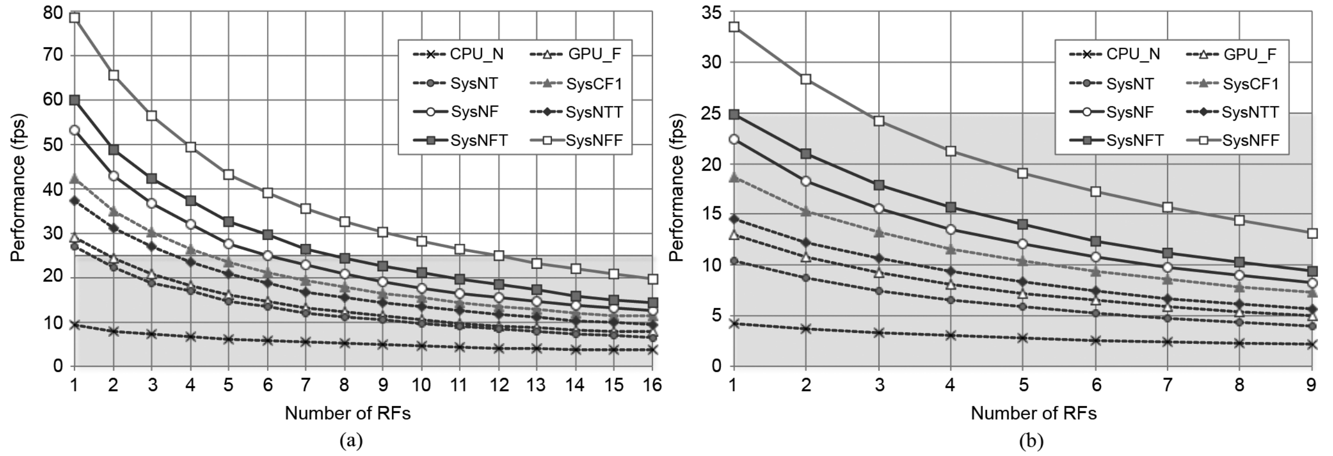


Fig. 5. Achieved performance (in fps) for different number of RF (SA size 32 × 32). (a) 1280 × 720 video resolution. (b) 1920 × 1080 video resolution.

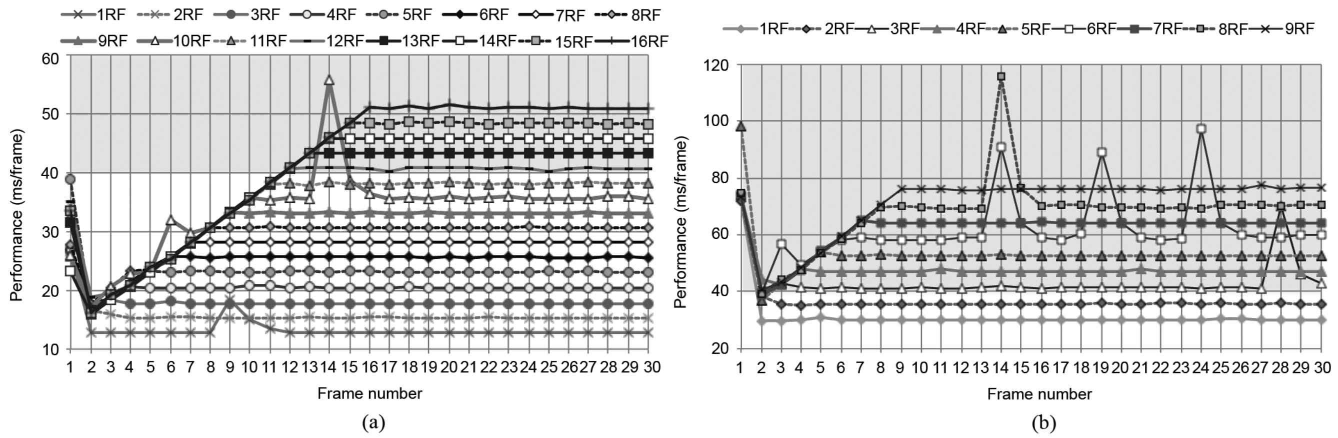


Fig. 6. Processing time per frame achieved with the proposed method for the first 30 inter-frames on *SysNFF* platform. (a) 1280 × 720 video resolution. (b) 1920 × 1080 video resolution.

TABLE III
PERFORMANCE COMPARISON (IN FPS) CONSIDERING DIFFERENT PLATFORMS FOR 3840 × 2160 RESOLUTION (UHD).

Method	Proposed		GPU only	CPU only
Platform	SysNFF	SysNF	GPU_F	CPU_N
1 RF	10.03	6.91	4.35	1.11
2 RF	8.01	5.30	3.37	0.91

during the execution. For example, on *SysNT* the MC, T&Q and $(\mathrm{T\&Q})^{-1}$ are typically assigned to the *GPU_T*, while the DBL module is rather mapped to the *CPU_N*. On the other hand, for the systems with more powerful GPUs, e.g., *SysNF* or *SysNFF*, all the MC, T&Q, $(\mathrm{T\&Q})^{-1}$ and DBL modules are usually mapped to the *GPU_F*. However, the impact of these modules to the overall encoding time is very small [39].

The applicability of the *proposed* approach was also assessed for future video resolutions, such as 3840 × 2160 pixels Ultra High Definition (UHD). The obtained results for UHD are presented in Table III and show that the proposed method is capable of preserving the speedups obtained for the HD resolutions, thus indicating the algorithm's scalability regarding both the number of RF and the video resolution.

Fig. 6 illustrates the capability of the *proposed* algorithm to provide load balancing decisions in non-dedicated systems, where the performance might vary along the time, depending on the state of the platform or devices (e.g., when the system resources are not used in exclusivity). For such purpose, the presented performance (in time per frame) was obtained by applying the *proposed* algorithm when encoding the first 30 inter-frames of a video sequence, immediately after the leading intra-frame (frame 0). The results are presented for both 1280 × 720 and 1920 × 1080 video resolutions and for different number of RFs. As described in Section IV, the *Initialization phase* of the *proposed* algorithm was applied in the encoding of the first inter-frame, in order to obtain the initial device/module performance parameters. As a consequence, the performance obtained for frame 1 in Fig. 6 corresponds to the processing time with the equidistant load partitioning for simultaneous collaborative processing and initial assessment of device/module execution affinities. Then, by applying the *Iterative phase* of the algorithm, significant performance improvements can be observed when subsequent inter-frames are encoded, from frame 2. The raising slopes at the beginning of this *Iterative phase* correspond to a temporary limitation of the parametrized inter-frame prediction mechanism. In fact, in order to invoke the encoding procedure with a desired number of RFs, it is required the completion of the processing of a number of frames equal or greater than the specified number of RFs. Due to the ability to iteratively assess and improve the device performance parameters, it is also observed that the presented approach is capable of providing load balancing, despite this gradual increase in the computational load. Once the

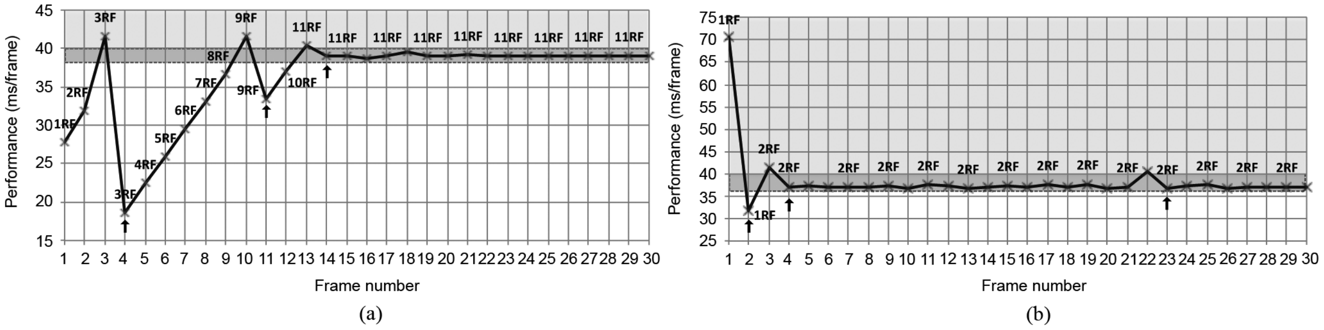


Fig. 7. Processing time per frame achieved with the proposed adaptive algorithm on *SysNFF* platform. (a) 1280 × 720 video resolution. (b) 1920 × 1080 video resolution.

predefined number of RFs is reached, the *proposed* algorithm converges towards very stable load balancing solutions, which is evidenced with the constant performance after this initial period (e.g., for 2, 4, 5, 7 and 9 RFs).

An interesting phenomenon worth noting was observed during the encoding of the 1280×720 sequence for 1 RF (frame 9) and 10 RFs (frames 6 and 14), as well as in the 1920×1080 sequence for 3 RFs (frame 28), 6 RFs (frames 3, 14, 19 and 24) and 8 RFs (frame 14), where a sudden change in the system performance has occurred (e.g., other processes started running). Still, the dynamic performance detection of the *proposed* algorithm allowed to capture this unexpected performance change, resulting in a successful load redistribution convergence according to the new state of the platform. This is emphasized by a relatively fast recovery of the performance curves, which only required a few inter-frames to converge to the regions with stable load balancing decisions. This ability of the *proposed* algorithm to provide stable distributions despite sudden system performance changes highlights the self-adaptability characteristics of the presented approach.

It is also worth emphasizing that during the above-presented experimental evaluation, the light-weight scheduling operations in the *proposed* algorithm take, in average, less than 1ms per inter-frame encoding, which is significantly less than the time required to individually execute any inter-loop module.

Finally, to further show the practical advantages of the proposed approach, the achieved encoding speed and RD performance were compared with the widely-used x264 encoder [46]. In contrast to the proposed procedure, which considers a collaborative inter-loop encoding on multi-core CPU and multi-GPU environments, the x264 encoder only supports the execution on multi-core CPU platforms. Hence, in order to provide a fair comparison between these two approaches, a series of tests was conducted on a multi-core CPU (*CPU_N*) from the *SysNFF* platform with different video sequences and resolutions, under the same testing conditions and similar encoding presets (e.g., by performing the "traditional" FSBM-based ME). As expected, x264 achieves RD performance similar to those that are obtained by encoding with the original FSBM algorithm from JM software (see Table V-A and [39]), and an encoding speed similar to the one obtained on *CPU_N* with the proposed approach, e.g., it achieves around 6fps for 720p (4fps for 1080p) for 4 RFs and the same conditions as in Fig. 5. However, as it was previously referred, by exploiting the full *SysNFF* capabilities for collaborative CPU+GPU processing, it is possible to achieve significant speed-ups when compared to the encoding on the multi-core *CPU_N* (up to 8.5) with the approach proposed herein.

C. Automatic Tuning of the Encoding Parameters for Real-Time Processing

The dynamic nature of the proposed load balancing method opens the possibility to adapt the video coding parametrization according to the targeted encoding performance. As an example, some coding parameters, such as the number of RF, can be adaptively adjusted in order to comply with other predefined encoding goals or even streaming bandwidth limitations. In particular, the proposed procedure to automatically tune the number of RFs may also provide the minimization of the scheduling overhead, by reducing the number of algorithm's invocations. In detail, determined load balancing distributions obtained for previously encoded frames can be re-used for subsequent inter-frame encodings (with the increased number of RFs) as long as the real-time processing is attained. Once the processing does not achieve the real-time encoding, the performance parameters are updated and the *Iterative phase* of the *proposed* algorithm is invoked.

Fig. 7 illustrates the implemented procedure to determine the upper bound on the number of RFs (up to a maximum of 16 RF, as defined by the H.264/AVC standard), in order to guarantee a real-time video encoding (≈ 25 fps), i.e., around 40 ms per frame). In order to deal with small performance variations in non-dedicated systems, minimum and maximum threshold values were predefined (below the hard limit for real-time encoding of 40 ms per frame), which define the region of realistically attainable real-time processing (dark gray regions in Fig. 7). The *proposed* algorithm was initially invoked for 1 RF and the obtained load distribution was re-used for the encoding of subsequent inter-frames, with an increased number of RFs. For the case of the 1280 × 720 video sequence, the initially obtained load distribution was re-used during the encoding of the first three inter-frames. However, in the third inter-frame, the re-used cross-device distribution no longer allowed to achieve real-time processing, thus resulting in an update of the module-specific device parameters and in a new invocation of the proposed algorithm to determine the new load balancing solution (see arrow in Fig. 7). This is emphasized by the improved performance when encoding the forth inter-frame with 3 RF, for which the real-time processing was achieved. The encoding of subsequent inter-frames was performed by increasing the number of RFs (until 9 RFs) and by reusing the obtained load balancing solution. Since the predefined threshold limit was no longer achieved when re-using the solution in the 10th inter-frame, the *proposed* algorithm had to be invoked again, which allowed the real-time processing with 9 RFs in the 11th inter-frame. This procedure continues until reaching 11 RFs, which marks the steady load balancing state with a maximum number of RFs that allows the real-time encoding. A similar behavior can be observed for the 1920×1080 resolution, where the real-time encoding was automatically found for 2 RFs within the interval of the first 30 inter-frames.

VI. Conclusions and Future Works

The presented contributions of this manuscript were focused on achieving real-time inter-loop encoding of HD video sequences, by simultaneous executing the encoding modules in heterogeneous CPU+GPU platforms. Efficient parallelization methods for all H.264/AVC inter-loop modules on both multi-core CPU and GPU architectures were investigated. Furthermore, an adaptive load balancing algorithm that efficiently distributes the loads across different heterogeneous devices, in order to allow a collaborative multi-module processing for ME+INT and SME with negligible scheduling overheads was proposed. For the remaining inter-loop modules, it was adopted a scheduling strategy that relies on module-device execution affinities, determined during the run-time. By dynamically capturing the module-specific performance parameters for each device, the presented approach was capable of iteratively improving the load balancing decisions, by adapting to the performance changes in non-dedicated systems.

The proposed approach was experimentally evaluated on different heterogeneous platforms with multi-core CPUs and multi-GPUs, for various HD video resolutions (i.e., 1280×720 and 1920×1080 pixels), as well as for UHD resolution (3840×2160 pixels). The efficiency of the proposed CPU/GPU parallelization methods was extensively assessed by considering the RD performance and scalability over both SA size and number of RFs. For all tested resolutions and platforms, the proposed algorithm was capable of delivering a performance which was several times higher than the one obtained by using a single-device. The self-adaptable characteristics of the proposed algorithm also allowed to automatically tune certain encoding parameters (e.g., the number of RFs in ME) to meet real-time constraints, converging to the maximum achievable improvement in what concerns the quality of the reconstructed video.

The future developments of the presented research will focus on efficient CPU/GPU parallelization of a complete encoder that also includes block intra-prediction, B-frames and entropy coding modules. New possibilities to further validate and improve the proposed scheduling/load balancing strategies for CPU+GPU systems with even higher degree of heterogeneity will be further investigated, in order to achieve real-time video encoding for a higher number of RFs, SA sizes and (U)HD resolutions.

References

[1] J. Ostermann *et al.*, "Video coding with H.264/AVC: tools, performance, and complexity," *IEEE Circuits Syst. Mag.*, vol. 4, no. 4, pp. 7–28, 2004.

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

[3] T. Wiegand, H. Schwarz, A. Joch, F. Kossentini, and G. J. Sullivan, "Rate-constrained coder control and comparison of video coding standards," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 688–703, 2003.

[4] OpenCL Specification v1.2, Khronos OpenCL Working Group, 2011.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[6] L. Zhang, J. Wang, C. Chu, and X. Ji, "Parallel motion compensation interpolation in H.264/AVC using graphic processing units," in *Proc. Int. Conf. Ind. Control and Electronics Eng. (ICICEE)*, 2012, pp. 767–771.

[7] B. Pieters, D. V. Rijsselbergen, W. D. Neve, and R. V. d. Walle, "Motion compensation and reconstruction of H.264/AVC video bitstreams using the GPU," in *Proc. Int. Workshop Image Anal. for Multimedia Interactive Services*, 2007, p. 69.

[8] A. Obukhov and A. Kharlamovl, Discrete Cosine Transform for 8x8 Blocks With CUDA, "NVIDIA," Research Report 2008.

[9] B. Wang, M. Alvarez-Mesa, C. Chi, and B. Juurlink, "An Optimized Parallel IDCT on Graphics Processing Units," in *Proc. Euro-Par 2012: Workshops, Springer*, 2013, vol. 7640, pp. 155–164.

[10] B. Pieters *et al.*, "Parallel deblocking filtering in MPEG-4 AVC/H.264 on massively parallel architectures," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 21, no. 1, pp. 96–100, 2011.

[11] J. Zhang, J. F. Nezan, and J.-G. Cousin, "Implementation of motion estimation based on heterogeneous parallel computing system with OpenCL," in *Proc. IEEE Int. Conf. High Performance Computing and Commun.*, 2012, pp. 41–45.

[12] S. Momcilovic and L. Sousa, "Development and evaluation of scalable video motion estimators on GPU," in *Proc. IEEE Workshop Signal Processing Syst. (SiPS)*, Oct. 2009, pp. 291–296.

[13] W.-N. Chen and H.-M. Hang, "H.264/AVC motion estimation implementation on compute unified device architecture (CUDA)," in *Proc. IEEE Int. Conf. on Multimedia & Expo (ICME)*, 2008, pp. 697–700.

[14] Z. Chen, J. Ji, and R. Li, "Asynchronous parallel computing model of global motion estimation with CUDA," *J. Comput.*, vol. 7, no. 2, pp. 341–348, 2012.

[15] R. Rodríguez-Sánchez, J. L. Martínez, G. Fernández-Escribano, J. L. Sánchez, and J. M. Claver, "A fast GPU-based motion estimation algorithm for H.264/AVC," *Advances in Multimedia Modeling, Springer*, pp. 551–562, 2012.

[16] Y. Gao and J. Zhou, "Motion vector extrapolation for parallel motion estimation on GPU," *Multimedia Tools Applicat.*, pp. 1–15, 2012.

[17] D. Chen, H. Su, W. Mei, L. Wang, and C. Zhang, "Scalable parallel motion estimation on muti-GPU system," in *Proc. Int. Symp. Comput., Commun., Control and Automation (ISCCCA)*, 2013, p. 6.

[18] B. Pieters, C. F. Hollemeersch, P. Lambert, and R. V. D. Walle, "Motion estimation for H.264/AVC on multiple GPUs using NVIDIA CUDA," *Proc. Soc. Photo-Optical Instrumentation Eng.*, vol. 7443, pp. 12–12, 2009.

[19] N.-M. Cheung, X. Fan, O. C. Au, and M.-C. Kung, "Video coding on multicore graphics processors," *IEEE Signal Process. Mag.*, vol. 27, no. 2, pp. 79–89, 2010.

[20] M. Schwalb, R. Ewerth, and B. Freisleben, "Fast motion estimation on graphics hardware for H.264 video encoding," *IEEE Trans. Multimedia*, vol. 11, no. 1, pp. 1–10, 2009.

[21] Y. Ko, Y. Yi, and S. Ha, "An efficient parallelization technique for x264 encoder on heterogeneous platforms consisting of CPUs and GPUs," *J. Real-Time Image Process.*, pp. 1–14, 2013.

[22] H. Wei, J. Yu, and J. Li, "The design and evaluation of hierarchical multi-level parallelisms for H.264 encoder on multi-core architecture," *Comput. Sci. Inf. Syst.*, vol. 7, no. 1, pp. 189–200, 2010.

[23] N. Wu, M. Wen, J. Ren, H. Su, and D. Huang, "High-performance implementation of stream model based H.264 video coding on parallel processors," *Multimedia and Signal Processing, Springer*, vol. 346, pp. 420–427, 2012.

[24] Y.-L. Huang, Y.-C. Shen, and J.-L. Wu, "Scalable computation for spatially scalable video coding using NVIDIA CUDA and multi-core CPU," in *Proc. ACM Int. Conf. Multimedia (ACM MM)*, 2009, pp. 361–370.

[25] S. Momcilovic, N. Roma, and L. Sousa, "Multi-level parallelization of advanced video coding on hybrid CPU + GPU platforms," in *Proc. Euro-Par Workshops*, 2012, pp. 165–174.

[26] S. Momcilovic, N. Roma, and L. Sousa, "Exploiting task and data parallelism for advanced video coding on hybrid CPU+GPU platforms," *J. Real-Time Image Process.*, pp. 1–17, 2013.

[27] B. Veeravalli, D. Ghose, and T. G. Robertazzi, "Divisible load theory: A new paradigm for load scheduling in distributed systems," *Cluster Comput.*, vol. 6, pp. 7–17, 2003.

[28] L. Marchal, Y. Yang, H. Casanova, and Y. Robert, "Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms," *Int. J. High Perform. Comput. Applicat.*, vol. 20, no. 3, pp. 365–381, 2006.

[29] A. Ilic and L. Sousa, "Scheduling divisible loads on heterogeneous desktop systems with limited memory," in *Proc. Euro-Par Workshops*, 2012, pp. 491–501.

[30] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Proc. IEEE Int. Symp. Parallel and Distributed Processing with Applicat. (ISPA)*, 2012, pp. 683–690.

[31] G. Barlas, A. Hassan, and Y. A. Jundi, "An analytical approach to the design of parallel block cipher encryption/decryption: A CPU/GPU case study," in *Proc. Int. Conf. Parallel, Distributed, and Network-Based Processing (PDP)*, 2011, pp. 247–251.

[32] A. Lastovetsky and R. Reddy, "Distributed data partitioning for heterogeneous processors based on partial estimation of their functional performance models," *Proc. Euro-Par*, pp. 91–101, 2010.

[33] P. Li, B. Veeravalli, and A. Kassim, "Design and implementation of parallel video encoding strategies using divisible load analysis," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 15, no. 9, pp. 1098–1112, 2005.

[34] C.-Y. Lee *et al.*, "Multi-pass and frame parallel algorithms of motion estimation in H.264/AVC for generic GPU," in *Proc. IEEE Int. Conf. Multimedia & Expo*, 2007, pp. 1603–1606.

[35] A. M. Tourapis, "Enhanced predictive zonal search for single and multiple frame motion estimation," *Visual Commun. Image Process.*, pp. 1069–1079, 2002.

[36] Z. Chen, J. Xu, Y. He, and J. Zheng, "Fast integer-pel and fractional-pel motion estimation for H.264/AVC," *J. Visual Commun. Image Represent.*, pp. 264–290, 2005.

[37] M. Garland *et al.*, "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul. 2008.

[38] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2007, pp. 407–420.

[39] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa, Advanced Video Coding on CPUS and GPUS: Parallelization and RD Analysis, Inesc-id, Technical Report, 2013.

[40] H. F. A. Altunbasak, "Sad reuse in hierarchical motion estimation for the H.264 encoder," in *Proc. IEEE Int. Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, 2005, vol. 2, pp. 905–908.

[41] T. Tan, G. Sullivan, and T. Wedi, "Recommended simulation common conditions for coding efficiency experiments—rev. 3," *ITU-Telecommun. Standardization Sector, VCEG, Doc. VCEG-AI10*, Jul. 2008.

[42] Z. Zhao and P. Liang, "Data partition for wavefront parallelization of H.264 video encoder," in *Proc. Int. Symp. Circuits and Syst. (ISCAS)*, 2006.

[43] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[44] A. M. Tourapis, A. Leontaris, K. Suehring, and G. Sullivan, "H.264/MPEG-4 AVC Reference Software Manual," *Joint Video Team of ISO/IEC MPEG and ITU-T VCEG, JVT-AD010*, Jan. 2009.

[45] G. Bjøntegaard, "Calculation of average PSNR differences between RD curves," *ITU-Telecommun. Standardization Sector, VCEG, Doc. VCEG-M33*, Apr. 2001.

[46] Reference software x264, 2013. [Online]. Available: http://www.videolan.org/developers/x264.html.

Svetislav Momcilovic is a senior researcher at INESC-ID, Lisbon, Portugal, working in the area of parallel video coding on multicore architectures. He received his PhD in Electrical and Computer Engineering in 2011 from the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Portugal, where he worked under the supervision of Prof. Leonel Sousa. His current research interests are mainly focused on parallel computing, multicore architectures, heterogeneous parallel systems and video coding, including high efficiency, advanced, multiview and distributed video coding.

Aleksandar Ilic is currently pursuing his PhD thesis at the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Portugal and as a member of the SiPS research group at INESC-ID, under the supervision of Prof. Leonel Sousa. He obtained his MSc degree in Electrical and Computer Engineering in 2007 from the Faculty of Electrical Engineering of Nis, Serbia. His current research interests are mainly focused on efficient techniques for dynamic load balancing, task scheduling and runtime performance modeling in highly heterogeneous parallel systems.

Nuno Roma received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal, in 2008. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering of IST and a Senior Researcher of the Signal Processing Systems Group (SiPS) of Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research interests include specialized computer architectures for digital signal processing, parallel processing and high-performance computing. He has contributed to more than 50 papers to journals and international conferences. Dr. Roma is a Senior Member of the IEEE Circuits and Systems Society and a Member of ACM.

Leonel Sousa got the Ph.D. degree in electrical and computer engineering from the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Portugal, in 1996. He is currently a full professor in the Electrical and Computer Engineering Department at IST and a senior researcher at INESC-ID. His research interests include VLSI architectures, signal processing systems, computer architecture, and parallel computing. He has contributed more than 200 papers to international journals and conferences. He is an associate editor of the Eurasip Journal on Embedded Systems, and has served in the program committee of more than 100 international conferences and symposiums. He is currently a member of the HiPEAC and the ComplexHPC European Networks. He is a Fellow of the IET and a senior member of both the IEEE and the ACM.