# FEVES: Framework for Efficient Parallel Video Encoding on Heterogeneous Systems

Aleksandar Ilic, Svetislav Momcilovic, Nuno Roma and Leonel Sousa
INESC-ID/IST, Universidade de Lisboa
Rua Alves Redol, 9
1000-029 Lisbon, Portugal
Email: {Aleksandar.Ilic, Svetislav.Momcilovic, Nuno.Roma, Leonel.Sousa}@inesc-id.pt

***Abstract*—Lead by high performance computing potential of modern heterogeneous desktop systems and predominance of video content in general applications, we propose herein an autonomous unified video encoding framework for hybrid multi-core CPU and multi-GPU platforms. To fully exploit the capabilities of these platforms, the proposed framework integrates simultaneous execution control, automatic data access management, and adaptive scheduling and load balancing strategies to deal with the overall complexity of the video encoding procedure. These strategies consider the collaborative inter-loop encoding as an unified optimization problem to efficiently exploit several levels of concurrency between computation and communication. To support a wide range of CPU and GPU architectures, a specific encoding library is developed with highly optimized algorithms for all inter-loop modules. The obtained experimental results show that the proposed framework allows achieving a real-time encoding of full high-definition sequences in the state-of-the-art CPU+GPU systems, by outperforming individual GPU and quad-core CPU executions for more than 2 and 5 times, respectively.**

## I. Introduction

Driven by the advances in the manufacturing technology and introduced architectural improvements, a remarkable increase in the processing capabilities of commodity computing systems can be evidenced in the past decade. This is especially notable for off-the-shelf desktop and servers systems, which currently do not only rely on multi-core Central Processing Units (CPUs) to perform general-purpose computations, but also employ different types of accelerators and co-processors to further enhance the overall computing power. Recently, Graphics Processing Units (GPUs) emerged as one of the most widely employed co-processors to the CPU, due to their intrinsic availability and possibility to extend their use beyond the traditional graphics purposes. In fact, even current and future trends in computer architecture move toward merging the functionally of these devices into a single chip, such as Intel Haswell or AMD Accelerated Processing Units (APUs).

Although heterogenous CPU+GPU systems are capable of delivering high computational throughput, exploiting their full potential is not a trivial task since it is required to explicitly deal with the complexity of both the overall system/devices and target applications. Firstly, the efficient employment of architecturally different devices requires the development of independent parallel algorithms optimized for each architecture by relying on several vendor-specific programming models, tools and libraries. Secondly, to simultaneously perform computations of a single application, these implementations need to be integrated in an unified execution environment. Although recent programming frameworks, such as OpenCL, aim at lessening these challenges by offering unified models applicable to a range of different architectures, this generalization usually comes at the cost of introducing additional execution overheads and inability to fully exploit architecture-specific features. In fact, these frameworks mainly tackle the platform programmability issues, which is only a part of the problem. In CPU+GPU systems, it is of the utmost importance to provide autonomous and adaptive mechanisms to guarantee the efficient use of all employed heterogeneous devices, inter-device communication links and other system resources. Therefore, the unified execution framework also needs to incorporate efficient scheduling and load balancing, as well as to provide a simplified interface and fulfill the overall application functionally, while hiding the system complexity from the end-user.

On the other hand, attaining the maximum system performance is also limited by specific characteristics and complexity of the target application. For example, highly data-parallel applications are usually better suited for GPU architecture with thousands of simple cores, while coarser-grained parallelism is often better exploited on multi-core CPUs. However, real-world applications usually consist of several different execution modules with diverse characteristics. During the execution, a complex interaction between these modules is often imposed due to the inherent data-dependencies, e.g., where the output of one module represents the input for another and/or simultaneous access to a single data buffer by several modules. As a result, the unified execution environment needs to consider both application- and architecture-specific characteristics, in order to achieve an as efficient as possible collaborative execution in heterogeneous environments. Due to the fact that it is generally impossible to consider an exhaustive set of characteristics, a full range of inter-module interactions and execution scenarios for general applications, a highly practical approach is herein proposed by focusing on a video encoding procedure to describe the application complexity.

Considering a clear dominance of video contents in the overall Internet traffic [1] and growing market demands for higher video resolutions, efficient video compression is absolutely essential to reduce the increased network bandwidth and data storage requirements. The newest video coding standards, such as H.264/AVC [2] and HEVC/H.265 [3], have established advanced coding methods to achieve higher compression rates, while retaining the video quality. However, such compression efficiency is payed by a dramatic increase in computing com-

plexity, specially on the encoder side, which makes it hard to be processed in real-time on any individual device available in current desktops. In the encoding procedure, a set of processing modules with diverse characteristics is iteratively applied on each video frame to reduce the spatial, temporal and statistical redundancy in video information. The resulting overall complexity involves several levels of data dependencies, namely: *i)* within a single module (spatial dependences within a frame); *ii)* across several modules (single-frame encoding); and *iii)* between different frames (temporal dependences between encoding iterations). Hence, all these dependencies make task and data parallelism hard to be exploited with straight-forward parallelization approaches.

In order to cope with such complexity and dependencies, we propose herein a robust FEVES framework for inter-loop video encoding that employs adaptive scheduling and load balancing techniques to challenge real-time encoding of High Definition (HD) sequences on multi-core CPU and multi-GPU platforms. To fully exploit the computing power of such heterogeneous platform, different parallelization strategies were specifically applied to each inter-loop video encoding module and for each device architecture, by relying on vendor-specific models and tools. These parallelized modules are integrated in the proposed unified framework that allows simultaneous execution control, automatic data access management, and cross-device workload distribution for different inter-loop video encoding modules. To the best of our knowledge, this is one of the first papers that throughly investigates efficient scheduling and load balancing methods for inter-loop video encoding on multi-core CPU and multi-GPU systems as an unified optimization problem, taking into account several concurrency levels between computation and communication.

The contributions of this paper are summarized as follows:

- unified video encoding framework with an automatic data access management for efficient orchestration of the inter-loop procedure in heterogeneous systems;
- scheduling and load balancing based on linear programming and realistic performance parametrization; these techniques explicitly consider: the complexity of the video encoding modules and the overall inter-loop procedure, performance disparity of heterogeneous devices, asymmetric bandwidth of communication links, supported amount of computation/communication concurrency, minimization of communication volume, and on-the-fly adaptation to the current state of the platform;
- highly optimized parallel video encoding modules for different generations of multi-core CPUs and GPUs;
- real-time H.264/AVC inter-loop video encoding of full HD sequences (>25fps) with Full-Search Block-Matching (FSBM) motion estimation and 4 Reference Frames (RFs); the implemented encoder is scalable over both the number of computing devices and the encoding parameters, e.g., RF and search area (SA).

## II. Background and Related work

According to the H.264/AVC standard [2], the Current Frame (CF) is divided in multiple square-shaped 16×16 (pixels) Macroblocks (MBs), which are encoded using either an intra- or an inter-prediction mode, as presented in Fig. 1. The most computationally demanding and frequently applied inter-prediction mode starts with the **Motion Estimation (ME)** module, which finds the best matching candidate for the currently processed MB within the predefined SAs in previously encoded RFs. In order to better suite to different shapes of moving objects, this standard allows further MB subdivisions according to 7 different partitioning modes, namely of 16×16, 16×8, 8×16, 8×8, 8×4, 4×8 and 4×4 pixels. The best matching candidate for each MB-partition is selected as the one with the minimum distortion value, which is computed according to the Sum of Absolute Difference (SAD) between all pixels from the current MB-partition and the examined candidates in the SA. The output of the ME are the offsets to the best matching candidates, i.e., motion vectors (MVs). In order to refine the ME, the RFs are additionally interpolated, by applying 6-tap and linear filters in the **Interpolation (INT)** module. The output of the INT module is stored in a Sub-pixel interpolated Frame (SF) structure, whose size is as large as 16 RFs. By relying on the preliminary MVs from the ME and on the interpolated SFs from the INT, the **Sub-Pixel Motion Estimation (SME)** is applied to further refine the MVs and provide a better prediction for each MB-partition. In total, the computations of these three modules takes about 90% of the overall video encoding time, both on CPU and GPU [4].

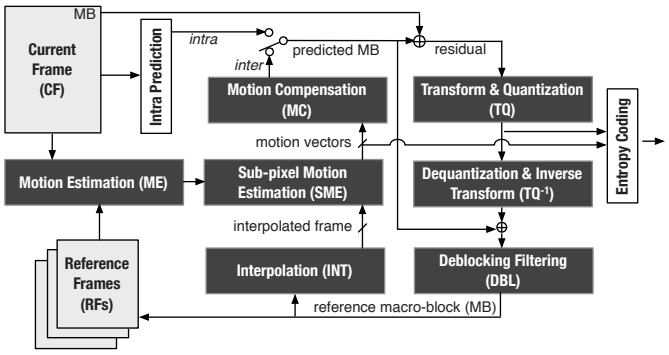



Fig. 1. Block diagram of the H.264/AVC encoder: inter-loop.

According to the adopted distortion metric and the refined MVs from the SME module, the best MB-partitioning mode is selected for each MB in the **Motion Compensation (MC)** module. For the selected mode, the prediction of each MB is computed, by subtracting the best matching SF candidates from the original MB in order to obtain the prediction *residual*. **Transform and Quantization (TQ)** are then applied to the *residual*, which is entropy coded and finally sent to the decoder. **Inverse Transform and Dequantization ($TQ^{-1}$)** are then applied to the quantized coefficients, to reconstruct the residual and reference MBs in the RFs, which are further used in subsequent ME procedures. Finally, to remove the blocking artifacts in the RF, the **Deblocking Filtering (DBL)** is applied on the MBs and MB-partitioning edges. Due to their lower share in the overall inter-loop video encoding [4], MC, TQ, $TQ^{-1}$ and DBL are herein referred as **R* modules**.

Focusing on state-of-the-art approaches, only rare attempts were made to efficiently parallelize the complete encoder (or its main functional parts) for CPU+GPU systems. The

adopted approaches usually *i)* simply offload one of the inter-loop modules in its entirety (mainly ME) to the GPU, while performing the rest of the encoder on the CPU [5]–[8], or *ii)* explore simultaneous CPU+GPU processing at the level of a single inter-loop module [9]. However, these approaches offer a limited scalability since only one GPU device can be efficiently employed. Furthermore, by offloading a single module the capabilities of a multi-core CPU are underused, by favoring execution on the GPU [5], [6], [8]. Moreover, for a single-module simultaneous CPU+GPU processing, the considered methods for cross-device load distribution usually perform an exhaustive search over the set of possible distributions and/or rely on simplified models for both module/platform performance. In detail, the optimal partitioning for "sub-frame" pipelining is decided through a large set of experiments in [7]; while a single-GPU and constant compute-only performance parametrization is used in [4]; whereas the load distribution in [9] is found by intersecting the fitted full performance curves for each device (experimentally obtained before module execution). Moreover, several works also consider the video encoding in homogeneous multi-GPU environments [10], where CPUs are not used for computing and an equidistant data partitioning of CF/RFs is applied, which do not allow an efficient load balancing in heterogenous environments.

In contrast, our recent contributions in the area of collaborative inter-loop video encoding consider the possibility of applying synchronous load balancing mechanisms at the level of individual modules, while focusing on the Rate-Distortion (RD) performance and efficient parallelization of inter-loop modules for CPU and GPU architectures [11]. The work presented herein greatly advances these contributions, by proposing an autonomous unified video encoding framework that integrates: *i*) improved implementations of the parallel inter-loop modules for modern device architectures (i.e., Intel Haswell CPU and NVIDIA Kepler GPU); *ii*) multi-level scheduling and load balancing methods that consider the complexity of the entire inter-loop when distributing the workloads among heterogeneous devices; *iii*) approaches for minimization of the communication volume and automatic data management in CPU+GPU platforms; and *iv*) adaptive execution methods that explicitly take into account the characteristics of heterogenous platform, such as performance disparity and amount of supported concurrency between the computation and data transfers. By relying on a row-based frame partitioning, the work proposed herein also tackles the issues related to efficient multi-application divisible load (DLT) scheduling, which is not yet completely covered in the literature [12]. Moreover, there are only a limited number of studies targeting the DLT scheduling in CPU+GPU systems even for general problems [13].

## III. FEVES: Framework for Inter-loop Video Encoding on Hybrid CPU+GPU Systems

In order to allow an efficient collaborative inter-loop video encoding on heterogeneous multi-core CPU and multi-GPU platforms, and to fully exploit the computational power of all heterogeneous devices, an unified execution framework is proposed herein, which integrates several blocks responsible for different framework functionalities (see Fig. 2). In particular, the key framework functionality is supported on the **Framework control** block, which interacts with all other main functional blocks, i.e., *Video Coding Manager*, *Load Balancing* and *Performance Characterization*. The **Video Coding Manager** is responsible for orchestrating the cross-device collaborative execution, by invoking the respective implementations of **Parallel Modules** on the available devices and for automatic **Data Access Management** between the system main memory (DRAM) and local device memories. The collaborative execution is conducted by relying on the workload distributions provided by the **Load Balancing** module, which is tightly coupled with an on-the-fly system and device **Performance Characterization**.

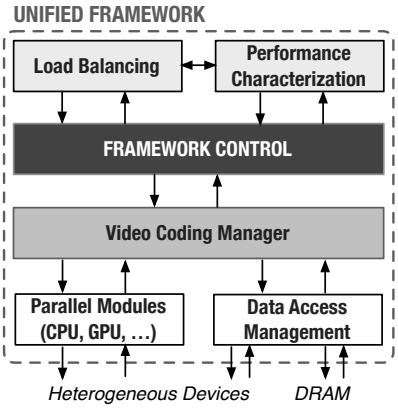



Fig. 2. Unified collaborative video encoding framework.

### A. Framework Control

As presented in Algorithm 1, the main functionality of the *Framework Control* considers two distinct phases, namely: *i*) the *initialization phase*, invoked for encoding the first inter-frame (lines 1-6); and *ii*) the *iterative phase*, applied during the processing of each subsequent inter-frame (lines 7-11).

In the **initialization phase**, the *Framework Control* firstly

**Algorithm 1** Framework Control (main functionality)

1: detect the number, type and characteristics of available devices
2: instantiate appropriate *Parallel Modules* implementations and configure *Video Coding Manager* and *Data Access Management*
3: call *Load Balancing* to determine initial equidistant partitioning for ME, INT and SME modules across all $p_i$ devices
4: invoke parallel execution and automatic transfers via *Video Coding Manager* and *Data Access Management*
5: record the execution and input/output data transfer times for each assigned load on $p_i$ device, as well as for remaining R* modules
6: perform intial *Performance Characterization*, by calculating per-device/module speeds and the asymmetric bandwidth of the interconnections for each non-CPU device
7: **for** $frame\_nr$=2 to $nr\_of\_inter\_frames$ **do**
8: call *Load Balancing* to determine the load distribution(s) based on *Performance Characterization*
9: invoke *Video Coding Manager*, *Data Access Management* and *Parallel Modules* to simultaneously process the assigned MB rows for computationally intensive modules on each $p_i$ device, as well as to process R* modules on the best processing device
10: record the corresponding times for computations and input/output transfers and update *Performance Characterization*
11: **end for**

instructs the detection of all available *heterogeneous devices* in the execution platform (line 1), which also includes the evaluation of architecture-specific capabilities of the available devices (e.g., support for different types of Single Instruction Multiple Data (SIMD) vector instructions), as well as the amount of supported concurrency between the kernel invocations and data transfers at the level of each accelerator (i.e., GPU). According to this information, the *Framework Control* instantiates the respective architecture-specific implementations of *Parallel Modules* for each available device, which are used during the collaborative encoding via the *Video Coding Manager* (line 2).

However, modern commodity CPU+GPU heterogeneous platforms incorporate a set of $n_c$ CPU cores and $n_w$ GPU accelerators, i.e., $p_i$ processing devices, where $i{=}\{1,\ldots,n_c{+}n_w\}$, as presented in Fig. 3. The depicted system resembles the fact that, in current desktop/server platforms, the accelerators are usually not stand-alone processing devices. Instead, they perform the computations on data fetched from the DRAM, where the CPU is responsible for initiating both on-device executions and data-transfers across the interconnection buses. Depending on the number of available communication engines, different GPU architectures support different amounts of concurrency between computation and communication, namely: for devices with a single copy engine, it is possible to overlap kernel executions with data transfers in a single direction (from CPU host to GPU device or vice versa); while for devices with dual copy engine it is additionally possible to overlap the transfers in opposite directions. Due to the fact that the approach proposed herein considers communication-aware scheduling and load balancing at the level of the complete inter-loop, identification of these accelerator properties is crucial to configure the *Video Coding Manager* and the *Data Access Management* blocks.

After configuring these functional blocks, the *Framework Control* invokes the *Load Balancing* routine (line 3 in Algorithm 1) to determine the workload distribution for each device-module execution pair, based on a realistic *Performance Characterization* of all important system resources. In particular, for each currently processed frame with $N$ MB rows, several distribution vectors are determined for the most computationally intensive modules, i.e, $m{=}\{m_i\}$ for ME, $l{=}\{l_i\}$ for INT and $s{=}\{s_i\}$ for SME, with the amount of MB rows to be processed on each heterogeneous $p_i$ device, such that $\sum_{i=1}^{n_w+n_c} m_i{=}\sum_{i=1}^{n_w+n_c} l_i{=}\sum_{i=1}^{n_w+n_c} s_i{=}N$. Since the proposed approach does not rely on any assumption regarding the performance of the devices, the interconnection links or other system resources, there are no *Performance Characterization* parameters that can be used when invoking the *Load Balancing* procedure for the first time. In order to build such initial performance estimations, the overall workload is equidistantly partitioned among all devices when processing the first frame. Hence, upon receiving the workload distributions, the *Framework Control* initiates the on-device executions and automatic data transfers via the *Video Coding Manager* (line 4). After processing the assigned equidistant distributions and remaining R* modules on each device, the respective execution and transfer times are recorded (line 5) and forwarded to the *Performance Characterization* (line 6).

In the **iterative phase**, when encoding each subsequent inter-frame, the *Framework Control* firstly invokes the *Load Balancing* routine, to determine the workload distributions for the most computationally intensive modules by taking into account the complete inter-loop procedure and the gathered *Performance Characterization* (line 8). By relying on the *Video Coding Manager* and *Data Access Management* configurations, all the *Parallel Modules* are collaboratively executed on each device (line 9), and the respective execution and transfer times are recoded and used to improve the accuracy of *Performance Characterization* (line 10).

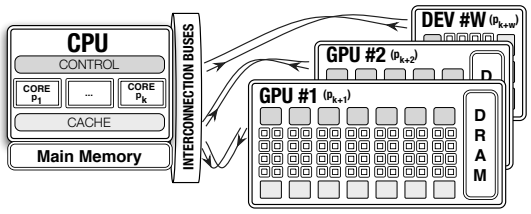



Fig. 3. Heterogeneous multi-core CPU and multi-GPU/accelerator system.

### B. Video Coding Manager

As previously referred, depending on the architecture-specific characteristics of the employed devices, the *Video Coding Manager* is responsible for orchestrating the collaborative video encoding in heterogeneous CPU+GPU systems, by invoking the execution of *Parallel Modules* and of the automatic *Data Access Management*. Hence, its functionality is defined during the initialization phase, according to the strict requirements that are imposed by the overall complexity of the video encoding procedure and platform characteristics.

In particular, the H.264/AVC standard (see Section II), the parallel execution at the entire inter-loop level brings to practice several hard-to-solve challenges. These challenges must be explicitly taken into account to ensure the correctness of the overall encoding procedure from the perspective of the sources of concurrency and the inherent data dependencies. In detail, an efficient parallelization requires the observance of data dependencies at several levels: *i)* between consecutive frames, *ii)* within a single video frame, and *iii)* between the encoding modules. In the H.264/AVC inter-prediction loop, the encoding of the CF can not start before the previous frames have been encoded and the required RFs have been reconstructed, which prevents the encoding of several frames in parallel. Moreover, the inherent data dependencies between the neighboring MBs in certain inter-loop modules (such as DBL) 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. Hence, all the data-dependent inter-loop modules have to be sequentially processed (within a single frame). The only exceptions are ME and INT, which can be simultaneously processed, since both of them use the CF and/or the RFs.

According to the performance of the respective CPU and

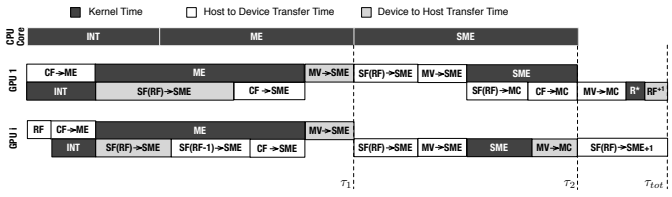


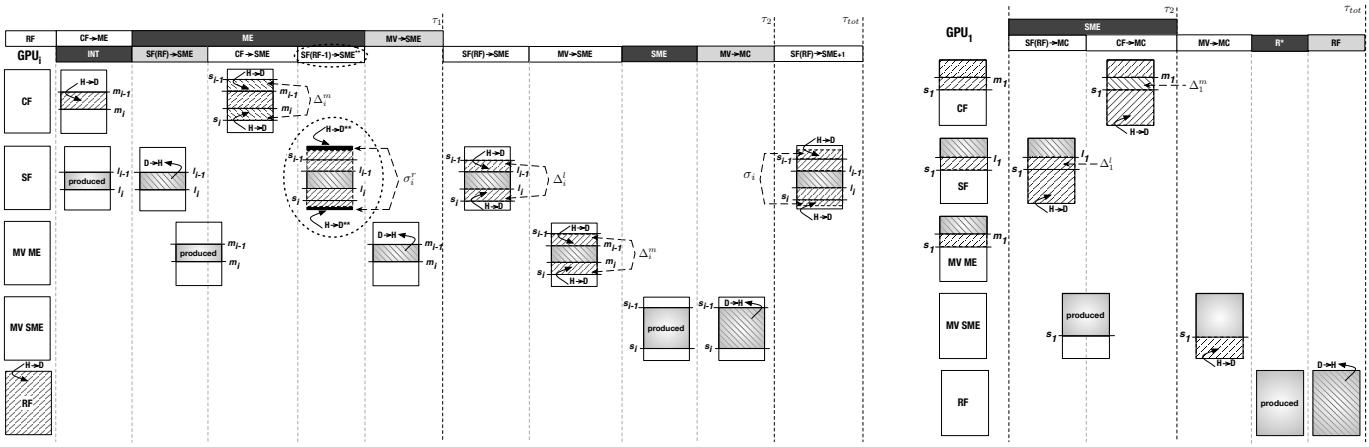

Fig. 4. GPU-centric configuration of the Video Coding Manager for heterogenous systems equipped with accelerators containing single-copy engines.

GPU parallel algorithms and inherent data dependencies [4], the inter-loop modules are divided in two groups, namely: *i*) ME, SME and INT modules; and *ii*) R* modules (i.e., MC, TQ, $TQ^{-1}$ and DBL). Due to the low computational complexity of MC, TQ and $TQ^{-1}$ modules (less than 3%) [4] and the limited possibility to collaboratively perform the DBL, the entire workload of the R* modules is assigned to a single (fastest) device, by applying the Dijkstra algorithm [11]. In fact, increased scheduling and communication overheads when performing cross-device load distribution for modules with such a low computational complexity would hardly compensate the expected performance gains, if any. In order to simplify the explanation of the overall functional principle behind the derived collaborative video encoding procedure, the single-device mapping of the entire R* encoding block will be considered. For example, if a single GPU device is assigned to perform all the R* modules, the derived procedure is designated as GPU-centric and the GPU device is marked as "selected accelerator", i.e., $GPU_1$. In contrast, a CPU-centric approach assumes that all the R* modules are performed on all CPU cores, in parallel.

As it was previously referred, the overall functionality of the *Video Coding Manager* also depends on the accelerator capabilities to sustain different amounts of concurrency between the computation and communication. Hence, the configuration of the *Video Coding Manager* differs on a per system basis, according to the identified accelerator capabilities in the initialization phase of the *Framework Control*. For example, Fig. 4 depicts the most common GPU-centric configuration of the *Video Coding Manager* for heterogenous systems equipped with accelerators containing single-copy engines, as well the corresponding orchestration of the *Parallel Modules* with the inevitable data transfers. As it can be observed, the *Video Coding Manager* is responsible for invoking the modules and data transfers in a particular order, such that the correctness of the overall encoding procedure is guaranteed. Hence, depending on the considered configuration, this module instantiates the parallel algorithms and maps them to several heterogenous devices, while specific input parameters (i.e., reference frames indexes) and parts of the frame required for the processing are supplied by the *Data Access Management*. Finally, the *Video Coding Manager* also provides the facilities to measure the execution and the transfer times, allowing *Performance Characterization* of the system resources.

According to the inherent data-dependencies that are imposed by the overall inter-loop video encoding procedure, several cross-device synchronization points are defined ($\tau_1$, $\tau_2$ and $\tau_{tot}$ points in Fig. 4). In brief, $\tau_1$ synchronization point reflects the dependency of the SME module on the outputs of ME and INT, while $\tau_2$ marks the completion of SME module and beginning of R* processing. It is worth emphasizing that the main objective of this framework is to minimize the total inter-loop video encoding time, i.e., $\tau_{tot}$. In detail, the main functionality of the *Video Coding Manager* is as follows:

- $\tau_1$ denotes the time when the assigned portions of the ME and INT modules are processed on each device (according to the determined $m_i$ and $l_i$ distributions from the *Load Balancing* block). This time also refers to the period when host to device transfers of the CF portions for ME (CF→ME) and for SME (CF→SME) are performed, as well as device to host transfers of the corresponding part of the interpolated SF (SF(RF)→SME) to ensure correctness of collaboratively processed SME. For accelerators not involved in the computation of the remaining R* modules ($GPU_i$), it is also required to fetch from the host the previously reconstructed RF before performing the ME and INT (RF), as well as to receive the remaining portion of the previously interpolated SF, SF(RF-1)→SME, to complete the SF at each accelerator. As expected, depending on the supported concurrency at the accelerator, these input and output data transfers are sequentially performed for devices with single-copy engine (see Fig. 4). In contrast, SF(RF)→SME device to host transfers can occur in parallel with the host to device CF→SME transfers for accelerators with dual-copy engine. Moreover,the device to host transfers of the computed MVs occur during this period (MV→SME), upon finishing the ME at each accelerator. It is worth noting that this approach also exploits the parallelism across independent modules (multiple divisible applications), i.e., ME and INT;
- $\tau_2$ represents the time when all heterogeneous devices should finish the collaborative processing of the SME module (according to the determined $s_i$ distributions). To sustain the SME parallel processing on the several accelerators, it starts by performing host to device transfers of the needed parts of the SF (SF(RF)→SME), as well as the missing MVs from ME (MV→SME), i.e., the needed MVs and SFs that were previously computed in the other devices. Moreover, the SME on all CPU cores can be started in this period, since all MVs are already present after the previous device to host transfers (performed in $\tau_1$). When a specific accelerator is selected to compute the remaining R* modules ($GPU_1$ in Fig. 4), the host to device transfers of the remaining parts of SF (SF→MC) and CF (CF→MC) are also performed to allow the correct execution of the MC module, in parallel with SME on the selected accelerator. For the other accelerators ($GPU_i$), it must be ensured that not only the computation of the SME portion is finished, but also that the computed MVs are transferred back to the host (MV→MC);
- $\tau_{tot}$ represents the overall inter-loop encoding time for a single inter-frame. In the period between $\tau_2$ and $\tau_{tot}$ the computation of the remaining R* modules needs to be finished, as well as the additional transfers of the remaining part of SF on the accelerators not selected to perform the R* modules (SF→SME+1). In the




(a) $GPU_i$ (b) $GPU_1$

Fig. 5. State of input/output buffers for accelerators with single-copy engine, when assigning $m_i$, $l_i$ and $s_i$ loads with the proposed load balancing approach.

case when an accelerator is selected to compute the remaining R* modules ($GPU_1$ in Fig. 4), prior to their computation, it is required to perform host to device transfers of the missing MVs from SME, which were computed on other devices (MV→MC). Furthermore, after R* are processed on $GPU_1$, the reconstructed RF ($RF^{+1}$) needs to be transferred back to the host, in order to allow its transfer at the beginning of the next inter-frame encoding on the other accelerators.

The following two subsections present further details regarding the implementation of *Parallel Modules*, as well as *Data Access Management* block.

*1) Parallel Modules:* During the orchestration of the video encoding procedure, the *Video Coding Manager* performs a mapping of the instantiated *Parallel Modules* to different devices available in the heterogeneous platform. In order to provide a wide support for different device architectures, an extensive library of highly optimized video encoding modules was specifically developed and made available, based on the different vendor-specific programming models and tools (i.e., OpenMP and SIMD intrinsics for multi-core CPUs, and CUDA/PTX for GPU devices). Parallel algorithms for CPUs do not only exploit the parallelism at the coarse-grained level (between the cores), but also the fine-grained data parallelism with SIMD instructions applied on successive memory locations within frame samples. In detail, different generations of multi-core CPUs, such as Intel Nehalem, Sandy/Ivy Bridge and Haswell micro-architectures, are supported by providing several different implementations of each module based on Streaming SIMD Extensions (SSE) 4.2, Advanced Vector Extensions (AVX) and AVX2 vector intrinsics. In addition, different per-module parallelizations are also developed for different generations of GPU architectures (i.e., NVIDIA Tesla, Fermi and Kepler) by exploiting fine-grained data parallelism with hundreds of GPU cores, as well by efficiently using the complex memory hierarchy. Hence, the adopted parallelization approaches highly vary not only for different device architectures, but also due to different computational loads of the encoding modules, inherent data dependencies, locality and regularity of data access patterns. Considering the limited space in this paper, and the complexity of the proposed framework, the description of the developed parallel algorithms is not provided herein and further details can be found in [14].

*2) Data Access Management:* As previously referred, the proposed framework also incapsulates specific mechanisms for device memory management and automatic data transfers in heterogeneous CPU+GPU systems via the *Data Access Management*. The functionality of this block is tightly coupled with the decisions taken by the *Load Balancing* routine. In order to better depict its major functional principles, Fig. 5 presents the state of the input and output buffers (i.e., CF, SF, RF, MVs from ME and SME), for different accelerator roles, different parts of the encoding procedure and each synchronization point from Fig. 4.

For the accelerators that are not responsible for computing of R* modules ($GPU_i$), the entire previously reconstructed RF is received in the first part of the $\tau_1$ time interval, followed by fetching of CF portion according to the given $m_i$ amount of loads to process. As it can be observed, the amount of transferred data is equal to $m_i \times MB\_height \times CF\_width$ and it must refer to the adequate CF position (offset) that is calculated with respect to the load distributions assigned to all other previously enumerated devices, which is symbolically referred to as $m_{i-1}$ in Fig. 5(a). At the same time, the SF part of size $l_i \times MB\_height \times SF\_width$ is interpolated and stored at the location calculated according to INT distributions assigned to other processing devices. The interpolated SF portion is then transferred to the host and the ME is performed on the previously received part of the CF to produce the $m_i \times MB\_height \times MV\_width$ fraction of correctly displaced MVs. At this stage, the remaining part of the previously interpolated SF (during the encoding of previous inter-frame) is transferred to the accelerator ($\sigma_i^r$). To ease the explanation of this procedure, it is presented in Fig. 5(a) as the SF remainder that is fetched in the next iteration (see dashed circled SF buffer, where the remaining portions are emphasized with dark solid colors). Moreover, during the $\tau_1$ time interval, additional CF portions are transferred to the host for SME. As it can be seen, depending on the computed CF offsets (in respect to the $s_i$ distributions for previously enumerated devices) the amount of data to be transferred varies. Hence, in the particular case depicted in Fig. 5(a), it is required to

perform two additional host to device transfers, to fetch the upper part of CF (region between $s_{i-1}$ and $m_{i-1}$) and its bottom part (region between $m_i$ and $s_i$). In order to reduce the overall communication volume, the proposed framework integrates a specific procedure that maximizes the reuse of already available data on devices. It takes into account the relative distance between distributions from different modules for the same device, as well as the offsets from the previously enumerated devices, in order to determine the amount of additional data to be transferred when the load distributions for different modules refer to the data located in the same buffer [15]. This procedure is explicitly considered in the *Load Balancing* block via MS_BOUNDS and LS_BOUNDS routines to determine the amount of additional transfers $\Delta_i^m$ and $\Delta_i^l$, respectively. Finally, at the end of $\tau_1$, the computed MVs from ME are transferred to the host.

At the beginning of the $\tau_2$ interval, the additional part of the SF for SME ($\Delta_i^l$) is received (represented as two data transfers between $l_{i-1}$ and $s_{i-1}$ and between $s_i$ and $l_i$ in Fig. 5(a). Similarly, additional MVs from ME (computed on other devices) are received ($\Delta_i^m$), which are represented in Fig. 5(a) as two separate data transfers (between $m_{i-1}$ and $s_{i-1}$ and between $s_i$ and $m_i$). Furthermore, $s_i \times MB\_height \times MV\_width$ MVs, computed by the SME, are stored at the appropriate positions and subsequently sent to the host. For the accelerator that is selected to perform the R* modules ($GPU_1$), the overall state of the input and output buffers does not significantly differ from the previously explained states for the other accelerators. However, as it is depicted in Fig. 5(b) at the end of the $\tau_2$ period for $GPU_1$, while the produced MVs in SME are stored, all remaining parts of SF and CF buffers are fetched from the host in order to allow the computation of MC (R*).

Finally, in the period between $\tau_2$ and $\tau_{tot}$, the remaining part of the SME MVs are also transferred from the host to the selected ($GPU_1$) accelerator, as presented in Fig. 5(b). This procedure is followed by the computation of R* modules, where the complete RF is produced and transferred back to the host, in order to allow a collaborative processing of the next inter-frame. During the same period, between $\tau_2$ and $\tau_{tot}$, the other accelerators ($GPU_i$) receive the remaining part of interpolated SF ($\sigma_i$), whose size is determined in order not to surpass $\tau_{tot}-\tau_2$ time limit (see Fig. 5(a)). This means that depending on the determined distributions and the amount of $\tau_{tot}-\tau_2$ time, the assigned time slot for additional SF transfers might not be sufficient to transfer all the needed data, i.e., the complete upper SF region (until $s_{i-1}$) and the complete bottom SF region (from $s_i$ to $N$), as depicted in Fig. 5(a). Therefore, the remaining (not transferred) part of SF must be received in the $\tau_1$ period, while encoding the next inter-frame (i.e., SF(RF$-1$)$\rightarrow$SME).

### C. Load Balancing and Performance Characterization

In order to efficiently exploit the computation power of the heterogeneous devices, asymmetric bandwidth of communication links and the computation/communication overlapping, while dealing with the overall complexity of the video encoding procedure and inherent data dependencies, a specific *Load Balancing* routine is herein proposed. In general, the proposed procedure differs according to the type of device that is selected to perform the remaining R* modules and

**Algorithm 2** Load Balancing for CPU+GPU video encoding

Input: $N, n_w, n_c, T_1^{R*}, K_i^m, K_i^l, K_i^s$
Input: $K_1^{rfdh}, K_i^{cfhd}, K_i^{rfhd}, K_i^{sfhd}, K_i^{sfdh}, K_i^{mvhd}, K_i^{mvdh}, \sigma_i^{r-1}$
Output: $m{=}\{m_i\}, l{=}\{l_i\}, s{=}\{s_i\}, \sigma{=}\{\sigma_i\}, \sigma^r{=}\{\sigma_i^r\}$
Objective: minimize $\tau_{tot}$

$$\sum_{i=1}^{n_w+n_c} m_i{=}N, \sum_{i=1}^{n_w+n_c} l_i{=}N, \sum_{i=1}^{n_w+n_c} s_i{=}N \quad (1)$$

$\forall i{\in}\{n_w{+}1, \ldots, n_w{+}n_c\}:$

$$l_i K_i^l + m_i K_i^m \leq \tau_1 \quad (2)$$

$$\tau_1 + s_i K_i^s \leq \tau_2 \quad (3)$$

$$m_1 K_1^{cfhd} + m_1 K_1^m + m_1 K_1^{mvdh} \leq \tau_1 \quad (4)$$

$$l_1 K_1^l + l_1 K_1^{sfdh} + \Delta_1^m K_1^{cfhd} + m_1 K_1^{mvdh} \leq \tau_1 \quad (5)$$

$$m_1 K_1^{cfhd} + l_1 K_1^{sfdh} + \Delta_1^m K_1^{cfhd} + m_1 K_1^{mvdh} \leq \tau_1 \quad (6)$$

$$\tau_1 + \Delta_1^l K_1^{sfhd} + \Delta_1^m K_1^{mvhd} + s_1 K_1^s \leq \tau_2 \quad (7)$$

$$\tau_1 + \Delta_1^l K_1^{sfhd} + \Delta_1^m K_1^{mvhd} + (N - l_1 - \Delta_1^l) K_1^{sfhd} + (N - m_1 - \Delta_1^m) K_1^{cfhd} \leq \tau_2 \quad (8)$$

$$\tau_2 + (N - s_1) K_1^{mvhd} + T_1^{R*} + N K_1^{rfdh} \leq \tau_{tot} \quad (9)$$

$\forall i{\in}\{2, \ldots, n_w\}:$

$$N K_i^{rfhd} + m_i K_i^{cfhd} + m_i K_i^m + m_i K_i^{mvdh} \leq \tau_1 \quad (10)$$

$$N K_i^{rfhd} + l_i K_i^l + l_i K_i^{sfdh} + \sigma_i^{r-1} K_i^{sfhd} + \Delta_i^m K_i^{cfhd} + m_i K_i^{mvdh} \leq \tau_1 \quad (11)$$

$$N K_i^{rfhd} + m_i K_i^{cfhd} + l_i K_i^{sfdh} + \sigma_i^{r-1} K_i^{sfhd} + \Delta_i^m K_i^{cfhd} + m_i K_i^{mvdh} \leq \tau_1 \quad (12)$$

$$\tau_1 + \Delta_i^l K_i^{sfhd} + \Delta_i^m K_i^{mvhd} + s_i K_i^s + s_i K_i^{mvdh} \leq \tau_2 \quad (13)$$

$$\sigma_i = \text{MIN}(N - l_i - \Delta_i^l, (\tau_{tot} - \tau_2)/K_i^{sfhd}) \quad (14)$$

$$\sigma_i^r = N - l_i - \Delta_i^l - \sigma_i \quad (15)$$

$\forall i{\in}\{1, \ldots, n_w\}:$

$$\Delta_i^m = \text{MS\_BOUNDS}(m, s) \quad (16)$$

$$\Delta_i^l = \text{LS\_BOUNDS}(l, s) \quad (17)$$

to the capabilities of the employed accelerators to support different amounts of concurrency between data transfers and kernel execution. Accordingly, Algorithm 2 describes the most commonly used GPU-centric variant of the *Load Balancing* routine for heterogeneous systems, with accelerators equipped with single-copy engine. The presented algorithm corresponds to the previously described configuration of the *Video Coding Manager* from Fig. 4, and to the functionality of the *Data Access Management*, depicted in Fig. 5.

The proposed *Load Balancing* procedure is based on linear programming and it allows the distribution of the loads of the most computationally demanding modules (i.e., ME, INT and SME) and mapping of the R* modules by relying on realistic *Performance Characterization* of the system resources. In brief, the proposed load balancing approach aimed at determining the distribution vectors for each computationally intensive module, i.e., $m{=}\{m_i\}$ for ME, $l{=}\{l_i\}$ for INT and $s{=}\{s_i\}$ for SME modules, with the amount of MB rows to be processed on each heterogeneous $p_i$ device and with the objective to minimize the total inter-loop video encoding time ($\tau_{tot}$). According to (1) from Algorithm 2, the sum of cross-device distributions for each module must be equal to the total number of MB rows ($N$) within the frame.

Moreover, two additional distribution vectors are also determined for each accelerator, i.e., $\sigma{=}\{\sigma_i\}$ and $\sigma^r{=}\{\sigma_i^r\}$, $i{=}\{1, \ldots, n_w\}$, to reflect the amount of additionally required data transfers (in MB rows) to complete the currently interpolated SF at each accelerator. Hence, the $\sigma{=}\{\sigma_i\}$ distribution vector reflects the amount of SF transfers that can be performed to each $p_i$ accelerator during the processing of the current inter-

frame, i.e., without causing any additional overheads to the total time for collaborative video encoding. Correspondingly, $\sigma^r=\{\sigma_i^r\}$ represents the remaining data transfers (in MB rows) to be performed at each $p_i$ accelerator to fully complete the SF, which occur during the processing of the next inter-frame. Hence, the computed $\sigma^r=\{\sigma_i^r\}$ remainders for the current frame serve as inputs when processing the next inter-frame, which are designated as $\sigma^{r-1}=\{\sigma_i^{r-1}\}$ vectors in Algorithm 2.

The performance of each $p_i$ device for the most computationally intensive modules is characterized with $K_i^m$, $K_i^l$, $K_i^s$ parameters for ME, INT and SME modules. These parameters are expressed in processing time per MB row, obtained for the currently assigned loads in $m$, $l$ and $s$ distribution vectors. Hence, the parameters obtained during the encoding with the already determined distributions serve as inputs to the *Load Balancing* procedure to determine the next distributions, according to the iterative phase from Algorithm 1. Correspondingly, $K_i^{cfhd}$, $K_i^{rfhd}$, $K_i^{rfdh}$, $K_i^{sfhd}$, $K_i^{sfdh}$, $K_i^{mvhd}$ and $K_i^{mvdh}$ represent the obtained time per transferred MB row in different directions, i.e., from host to device ($hd$) or from device to host ($dh$), for CF, RF, SF and MVs. Depending on the device that is selected to perform the computations of the remaining R* modules, $T_1^{R*}$ refers to the time required to perform the complete MC+TQ+TQ$^{-1}$+DBL sequence on the selected device (GPU$_1$). It is worth emphasizing that updating the *Performance Characterization* in runtime (after each processed frame) is particularly important for video coding on highly unreliable and non-dedicated systems, where the performance and available bandwidth can vary depending on the current state of the platform (e.g., load fluctuations, multi-user time sharing, operating system actions).

According to the previously presented analyses, the minimization of the total inter-loop video encoding time $\tau_{tot}$ is attained by satisfying different conditions from Algorithm 2, for different synchronization points and heterogeneous devices depicted in Fig. 4 and 5. Due to the limited space in this manuscript, the overall *Load Balancing* functionality can be briefly summarized as follows:

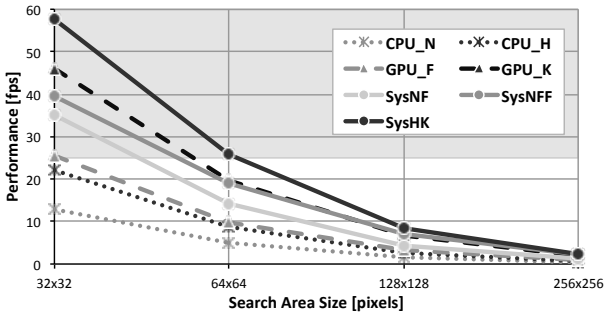
- (2) and (3) express the necessary conditions for multi-core CPU execution in respect to $\tau_1$ and $\tau_2$ synchronization points;
- (4)–(9) present the required conditions for the accelerator selected to perform the R* computations (GPU$_1$), which guarantee that all input and output transfers, as well as kernel executions, are accomplished according to the defined synchronization points in Fig. 4 and 5;
- (10)–(15) state the necessary conditions for the other employed accelerators (GPU$_i$) according to the execution scenarios presented in Fig. 4 and 5;
- (16) and (17) correspond to the previously referred procedures to determine the amount of additional transfers when two different modules share the access to a single data buffer, i.e., for ME and SME accessing the CF and MV buffers, as well as for INT and SME accessing the SF, respectively.

## IV. Experimental Results

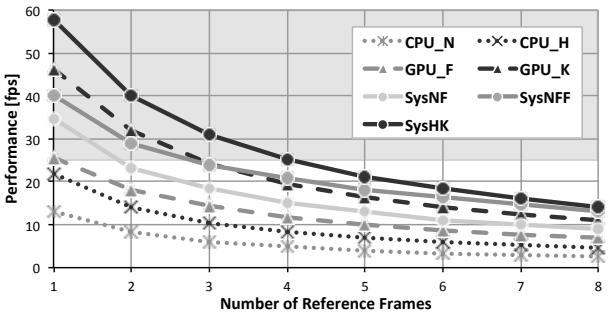
In order to evaluate the efficiency of the proposed framework and to challenge the real-time encoding for 1080p (full) HD video sequences on commodity desktop systems, the experimental evaluation was conducted on several different CPU and GPU device architectures, as well as on different configurations of CPU+GPU systems. The presented results were obtained by relying on a subset of optimized *Parallel Modules* for CPU and GPU architectures [11], [14] based on H.264/AVC JM 18.4 reference encoder. The tests were performed on OpenSUSE 13.1 operating system with CUDA 5.5, Intel Parallel Studio 12.1 and OpenMP 3.0.

Two different generations of quad-core CPU micro-architectures were considered: Intel Nehalem i7 950 (CPU_N) and the newest Intel Haswell i7 4770K (CPU_H) processors; while the GPU *Parallel Modules* were tested on two different NVIDIA architectures, namely: Fermi GTX 580 (GPU_F) and Kepler GTX 780 Ti (GPU_K). In terms of collaborative CPU+GPU processing environments, three different heterogeneous configurations were considered: *i*) SysNF, combining CPU_N and a single GPU_F; *ii*) SysNFF, with CPU_N and two GPU_F devices; and *iii*) SysHK, with CPU_H and a single GPU_K device. The proposed *Load Balancing* method was applied to encode different 1080p HD sequences ("*Toys and Calendar*" and "*Rolling Tomatoes*") by strictly following the VCEG recommendations [16] for IPPP sequences, Baseline Profile, and Quantization Parameter (QP) of {27,28} for {ISlice, PSlice}. The obtained video encoding performance was expressed in terms of the resulting encoding speed (i.e., time per frame or frames per second (fps)) and it does not significantly vary neither for different video sequences (due to adopted FSBM ME) nor for different QPs (due to low computational complexity of the R* modules).

In order to show the efficiency of the proposed framework when applying different video encoding parameters for different systems and device architectures, Fig. 6 presents the obtained experimental results when processing 1080p HD video sequences for four different SA sizes and different number of RFs. In all the presented charts, the shaded area represents the performance region where it is possible to achieve a real-time encoding. As it can be seen in Fig. 6(a), the overall performance of the inter-loop encoding significantly decreases between two successive SA sizes, due to the quadruplication of the ME computational load. It can also be observed that it is generally possible to obtain higher encoding performance by efficiently exploiting the improvements offered by the newest device architectures with optimized *Parallel Modules* (e.g., encoding on multi-core CPU_H is about 1.7 times faster than on CPU_N, while GPU_K outperforms GPU_F for almost 2 times in terms of the encoding speed). In fact, the efficiency of the implemented *Parallel Modules* can also be evidenced by the possibility of achieving real-time encoding ($\geq$25fps) for the smallest 32×32 SA size and 1 RF on both tested GPUs. On the other hand, for collaborative CPU+GPU systems, the proposed framework succeeded to efficiently exploit the synergetic performance of the heterogeneous devices for all the considered SA sizes by relying on the proposed *Load Balancing* strategy, thus significantly outperforming the corresponding single device executions. As a result, a real-time inter-loop encoding of 1080p sequences was achieved on all

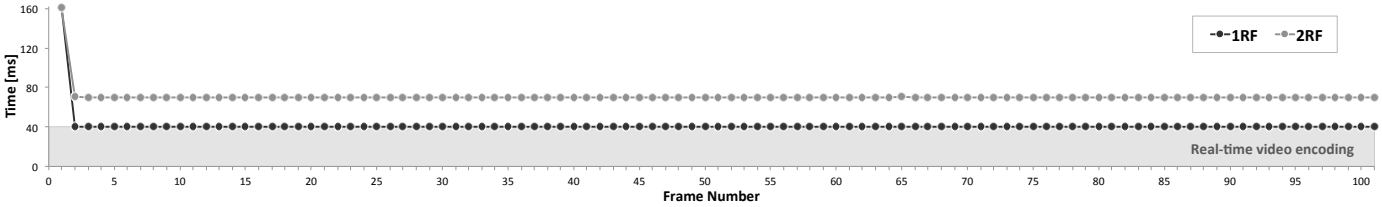



(a) Different SA sizes (1 RF)

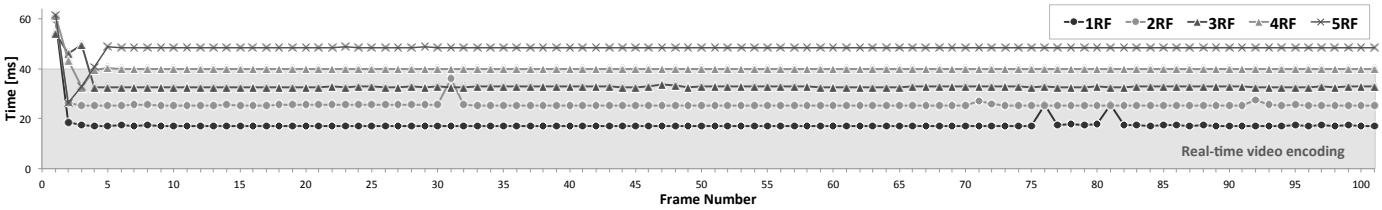



(b) Different number of RFs (32×32 SA).

Fig. 6. Performance obtained with the proposed framework for different CPU and GPU device architectures and CPU+GPU systems for 1080p sequences.




(a) Search area size of 64×64 pixels.




(b) Search area size of 32×32 pixels.

Fig. 7. Performance obtained with the proposed adaptive *Load Balancing* routine when encoding the first 100 frames of "Rolling Tomatoes" sequence.

tested CPU+GPU systems for the SA size of 32×32 and 1 RF. In particular, a real-time encoding was achieved even for a higher and more challenging 64×64 SA on SysHK, which was not attainable with the state-of-the-art approaches [11].

To further challenge the real-time inter-loop encoding on the off-the-shelf desktop systems, Fig. 6(b) presents the performance obtained with the proposed framework when encoding 1080p HD sequences for different number of RFs and SA size of 32×32. As it can be observed, the proposed *Load Balancing* strategy allowed achieving a real-time encoding on all tested CPU+GPU systems for multiple RFs. In particular, a real-time encoding was achieved for up to 4 RFs on SysHK platform, outperforming the execution on both SysNFF and SysNF platforms. Furthermore, by relying on the proposed strategy for all the considered number of RFs, an average speedup of about 1.3 was obtained in the SysHK platform when compared to the single GPU_K, and about 3 when compared to the multi-core CPU_H execution. On the SysNFF platform, speedups up to 2.2 and 5 were obtained when compared to the GPU_F and quad-core CPU_N execution, respectively.

In order to provide a better insight on the capabilities of the proposed framework to dynamically adapt the load balancing decisions according to the current state of the execution platform, Fig. 7 depicts the obtained encoding time when processing the fist 100 inter-frames of a 1080p video sequence in SysHK platform for different number of RFs and SA sizes, namely: SA of 64×64 in Fig. 7(a) and SA of 32×32 in Fig. 7(b). In both cases, the time obtained when encoding the first inter-frame corresponds to an equidistant workload partitioning applied in the initialization phase of the *Framework Control* (see Algorithm 1), in order to obtain the initial *Performance Characterization* parameters for relevant system resources, such as the performance of heterogeneous devices and asymmetric bandwidth of the communication links. After the initial characterization is performed (in the iterative phase of the *Framework Control*), the proposed *Load Balancing* routine (see Algorithm 2) is invoked in order to distribute the loads corresponding to the *Parallel Modules* among all heterogeneous devices. As it can be observed, by relying on the proposed strategy, a significant reduction in the encoding time can be observed when subsequent inter-frames are processed, starting already with frame 2. In particular, a real-time encoding was attained in both cases for different number of RFs, i.e., 1 RF for 64×64 SA and 4 RFs for 32×32 SA, which was not achievable with an equidistant partitioning.

Due to the higher computational complexity when encoding the inter-frames with larger SA size (64×64), the proposed strategy ensures a near-constant encoding time for all the processed inter-frames and considered RFs in Fig. 7(a). In the case of 32×32 SA size, the observed raising slopes in

Fig. 7(b), when the encoding is performed with the number of RFs higher than one (e.g., for 5 RFs), the encoding time is increasing with the number of encoded inter-frames at the beginning (frames 2-5), while it becomes near-constant after the fifth inter-frame. The rationale behind this behavior lies in the fact that a single RF is produced during the encoding of a single inter-frame (see Fig. 5). Hence, for encoding with a higher number of RFs, it is required to firstly process several inter-frames, in an amount that must be greater or equal to the specified number of RFs. During the processing of these initial inter-frames, the number of considered RFs increments with each processed inter-frame, until reaching the specified number of RFs. Therefore, Fig. 7(b) demonstrates the ability of the proposed *Load Balancing* strategy to efficiently distribute the loads and achieve a high encoding performance, while simultaneously dealing with this "on-the-fly" increasing problem complexity.

An interesting phenomenon was observed during the encoding with 1 RF (frames 76 and 81) and 2 RFs (frames 31, 71 and 92), where a sudden change in the system performance has occurred (e.g. other processes started running). Still, the dynamic *Performance Characterization* of the proposed framework allowed capturing this unexpected performance change, resulting in a successful load redistribution according to the new state of the platform. This is emphasized by a very fast recovery of the performance curves, which required a single inter-frame to converge to the regions with stable load balancing decisions. This ability of the proposed framework to provide stable distributions, despite sudden system performance changes, highlights the self-adaptability characteristics of the presented approach.

Finally, it is also worth emphasizing that the scheduling overheads introduced by the proposed framework take, on average, less than 2ms per inter-frame encoding, which is significantly less than the time required to individually execute any inter-loop module.

## V. Conclusions

In this paper, an autonomous video encoding FEVES framework for multi-core CPU and multi-GPU platforms is proposed. To challenge a real-time H.264/AVC inter-loop video encoding on these systems, adaptive scheduling and load balancing methods were integrated that explicitly take into account the overall complexity of the video encoding procedure, while efficiently exploiting several levels of concurrency between computation and communication. The support for a wide range of different CPU and GPU architectures is provided via specifically developed encoding library with highly optimized algorithms for all inter-loop modules. The proposed unified framework also incorporates simultaneous execution control across different heterogeneous devices and automatic data access management to minimize the overall communication volume. By relying on dynamic performance characterization for heterogeneous devices and communication links, the presented framework was capable of iteratively improving the load balancing decisions and adapting to the performance changes in non-dedicated systems. The efficiency of the proposed framework was experimentally evaluated in different CPU+GPU platforms, where a real-time video encoding of 1080p HD sequences was obtained even for more challenging video encoding parameters, such as 64×64 SA and/or multiple RFs. Finally, the FEVES framework was capable of outperforming a single-device execution for several times, while introducing negligible scheduling overheads.

## Acknowledgment

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under projects PEst-OE/EEI/LA0021/2013, PTDC/EEI-ELC/3152/2012 and PTDC/EEA-ELC/117329/2010.

## References

[1] Cisco, "Cisco visual networking index: Forecast and methodology, 2012-2017," White Paper, May 2013.

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

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

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

[5] 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. of ACM Int. Conf. on Multimedia*, 2009, pp. 361–370.

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

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

[8] 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," in *Advances in Multimedia Modeling*. Springer, 2012, pp. 551–562.

[9] J. Zhang, J. F. Nezan, and J.-G. Cousin, "Implementation of Motion Estimation Based on Heterogeneous Parallel Computing System with OpenCL," in *Proceedings of the IEEE Int. Conf. on High Perf. Comp. and Comm.*, 2012, pp. 41–45.

[10] B. Pieters, C. F. Hollemeersch, P. Lambert, and R. Van De Walle, "Motion estimation for H.264/AVC on multiple GPUs using NVIDIA CUDA," in *Proceedings of the Society Photo-Optical Instrumentation Engineers*, vol. 7443, 2009, p. 12.

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

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

[13] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Proceedings of the IEEE Int. Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2012, pp. 683–690.

[14] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa, "Advanced Video Coding on CPUs and GPUs: Parallelization and RD Analysis," INESC-ID, Technical Report (available online), February 2013.

[15] A. Ilic, S. Momcilovic, and L. Sousa, "Scheduling and Load Balancing for Multi-module Applications on Heterogeneous Systems," INESC-ID, Technical Report, February 2014. [Online]. Available: http://sips.inesc-id.pt/~ilic/publications/trch5.pdf

[16] T. Tan, G. Sullivan, and T. Wedi, "Recommended simulation common conditions for coding efficiency experiments - revision 3," *ITU, VCEG, Doc. VCEG-AI10*, July 2008.