Adaptive Scheduling Framework for Real-Time Video Encoding on Heterogeneous Systems

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

***Abstract*— To challenge real-time encoding of high-definition video sequences on heterogeneous desktop systems, a collaborative central processing units (CPU) + graphics processing unit (GPU) framework for interloop video encoding is proposed herein. The proposed framework considers the overall complexity of the collaborative interloop encoding as a unified optimization problem. Several functional blocks are integrated for simultaneous execution control, automatic data access management, performance characterization, and adaptive scheduling and load balancing. These blocks aim at fully exploiting the performance of heterogeneous devices, asymmetric bandwidth of communication links, and 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 interloop modules. The experimental results show that the proposed framework allows achieving a real-time encoding of full high-definition sequences in several CPU + GPU systems. It also delivers performance improvements of up to 61.2% over the state-of-the-art solution, while outperforming individual GPU and quad-core CPU executions by more than 2 and 5 times, respectively.**

***Index Terms*— General-purpose computation on graphics processing units, heterogeneous systems, load balancing, video coding.**

I. INTRODUCTION

GROWING market demands for higher video resolutions and a clear dominance of video content in the overall Internet traffic [1] lead to increased network bandwidth demands and data storage requirements. To cope with these issues, the newest video coding standards, such as H.264/Advanced Video Coding AVC [2] and High Efficiency Video Coding (HEVC)/H.265 [3], rely on advanced methods to achieve higher compression rates, while retaining the video quality. However, due to a dramatic increase in computing complexity, this compression efficiency is hard to be achieved in real time.

Modern desktop and server systems are capable of delivering a remarkable processing performance. In addition to multicore central processing units (CPUs), these systems also employ different accelerators/coprocessors for computation. Due to their intrinsic availability and high performance capabilities, graphics processing units (GPUs) emerged as one of the most widely used coprocessors. However, to fully exploit the CPU + GPU capabilities for efficient video encoding, it is required to cope with complexity of both system/device architecture and the overall encoding procedure.

In video encoding, several modules with diverse characteristics are iteratively applied to each frame to reduce the spatial, temporal, and statistical redundancies in video information. The resulting application complexity involves several levels of data dependencies, namely: 1) within a single module; 2) across several modules; and 3) between different frames. These inherent data dependencies impose a complex interaction between the encoding modules, e.g., where the output of one module represents the input for another and/or when several modules simultaneously access a single data buffer. Hence, all these dependencies make task and data parallelism hard to exploit with straightforward parallelization techniques.

For collaborative CPU + GPU processing, it is first required to develop and optimize independent parallel algorithms for each device architecture. For example, highly data-parallel algorithms usually better suit thousands of GPU cores, while coarser grained parallelism is often better exploited on multicore CPUs. Furthermore, the developed parallel implementations need to be integrated in a unified execution framework. This framework also needs to incorporate self-adaptive mechanisms to orchestrate the efficient use of employed heterogeneous devices, interdevice communication links, and other system resources. In addition, realistic characterization of the achievable performance and bandwidth across heterogeneous devices and communication links is of utmost importance to provide an efficient resource-aware system utilization.

To challenge real-time encoding of high definition (HD) sequences on heterogeneous desktop systems, we herein propose a collaborative CPU + GPU framework for interloop video encoding. The unified framework provides a simplified interface that fulfills the overall encoding functionally, while hiding the system complexity from the end user. It includes a library of highly optimized parallel interloop modules developed for different CPU and GPU architectures by relying on vendor-specific programming models and tools. The proposed framework also relies on specifically developed adaptive scheduling and load balancing to efficiently exploit

Manuscript received August 4, 2014; revised November 12, 2014 and January 10, 2015; accepted February 3, 2015. Date of publication February 11, 2015; date of current version March 3, 2016. This work was supported by the National Funds through the Fundação para a Ciência e a Tecnologia under Project UID/CEC/50021/2013 and Project PTDC/EEIELC/3152/2012. This paper was recommended by Associate Editor S.-Y. Chien.

The authors are with Instituto de Engenharia de Sistemas e Computadores—Investigação e Desenvolvimento, Instituto Superior Técnico, Universidade de Lisboa, Lisbon 1649-004, Portugal (e-mail: aleksandar.ilic@inesc-id.pt; svetislav.momcilovic@inesc-id.pt; nuno.roma@inesc-id.pt; leonel.sousa@inesc-id.pt).

Digital Object Identifier 10.1109/TCSVT.2015.2402893

1051-8215 © 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

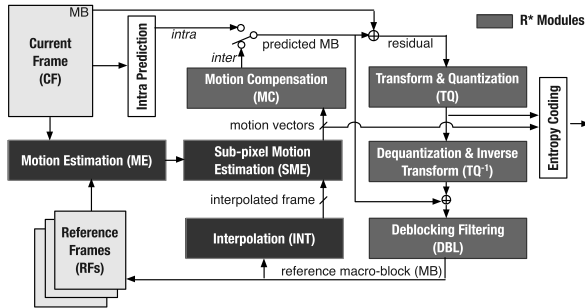

Fig. 1. Block diagram of the H.264/AVC encoder: interloop.

the collaborative performance of the underlying platform. In addition to cross-device workload distribution for different interloop modules, this framework also allows simultaneous execution control and automatic data access management. The contributions of this manuscript can be summarized as follows:

1) unified framework with automatic data management for efficient interloop encoding in heterogeneous systems;
2) auto-configuration according to the identified platform/device characteristics;
3) adaptive scheduling and load balancing based on linear programming and realistic performance parametrization;
4) communication minimization algorithm when two parallel modules share access to a single data buffer;
5) library of highly optimized parallel encoding modules for different generations of multicore CPUs and GPUs;
6) on-the-fly adaptation of encoding parameters according to the current state of the platform and the real-time limit;
7) real-time H.264/AVC interloop encoding of full HD sequences (>25 frames/s) with demanding encoding parameters.

To the best of our knowledge, there is no similar state-of-the-art solution that perceives the collaborative interloop encoding in heterogeneous environments as a unified optimization problem. Specifically, the proposed framework explicitly considers: 1) complexity of encoding modules and the overall interloop; 2) performance disparity of heterogeneous devices; 3) asymmetric bandwidth of communication links; 4) supported concurrency between the computation and communication; and 5) on-the-fly adaptation to the current state of the platform.

II. Background and Related Work

In the H.264/AVC standard [2], the current frame (CF) is divided into multiple 16×16 (pixels) macroblocks (MBs), which are encoded using either intra- or 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. In the ME, the best matching candidate for each MB is found within the predefined search areas (SAs) in the previously encoded reference frames (RFs). The offsets to the best matching candidates, i.e., motion vectors (MVs), represent the output of the ME. To better suit different shapes of moving objects, MB can be subdivided according to seven 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. This value is usually computed as the sum of the cost of the MV and the sum of absolute differences between all pixels from the current MB partition and the examined candidates in the SA. To refine the ME, the RFs are interpolated, by applying six-tap and linear filters in the interpolation (INT) module. The output of the INT module is stored in a subpixel interpolated frame (SF) structure, whose size is as large as 16 RFs. By relying on preliminary MVs from the ME and on interpolated SFs from the INT, the subpixel motion estimation (SME) is applied to further refine the MVs and provide a better prediction for each MB partition. For the video encoding parameters/configurations considered herein, the computation of these three modules takes about 90% of the overall encoding time, on both CPU and GPU [4], [5].

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 *residual* for each MB is computed by subtracting the best matching SF candidates from the original MB. Transform and quantization (TQ) are then applied to the *residual*, which is entropy coded and 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. Finally, to remove the blocking artifacts in the RF, deblocking filtering (DBL) is applied on the MBs and MB-partition edges. Due to their lower share in the interloop encoding time [4], [5], MC, transform and quantization (TQ), inverse transform and dequantization (TQ^{-1}), and DBL are referred herein as R* modules.

Focusing on the 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. These approaches usually offload a single interloop module in its entirety (mainly ME) to the GPU, while performing the rest of the encoder on the CPU [6]–[8]. Hence, they offer a limited scalability since only one GPU device can be efficiently employed. The multicore CPU capabilities are also underused while waiting for the offloaded module to be completed [7], [8]. Moreover, the approaches that distribute the load of a single module across CPU and GPU devices usually perform an exhaustive search over all possible distributions and/or rely on simplified module/device performance models. In detail, the load distribution is found: 1) by a large set of experiments for optimal subframe pipelining in [6]; 2) with constant compute-only performance parametrization in a single-GPU platform [4]; and 3) by intersecting the fitted full performance curves (experimentally obtained before module execution) for each device in the system [9]. Moreover, several works apply simplistic equidistant data partitioning of CF/RFs in homogeneous multi-GPU environments [10] without

considering CPUs for computing. In contrast, the framework proposed herein considers the complexity of the entire interloop as a unified optimization problem, when orchestrating the computation and communication from different modules among heterogeneous devices. It also integrates mechanisms for automatic data management, multilevel scheduling, and adaptive load balancing that rely on realistic performance characterization of devices and communication links.

The proposed framework also significantly advances our previous contributions in the area of collaborative interloop video encoding. In detail, the approach from [11] mainly tackles dynamic balancing of the *computational load* from two separate inter-prediction stages, i.e., ME + INT and SME. In contrast to the framework proposed herein, the approach from [11]: 1) is unaware of the overall interloop complexity and it does not consider data reuse among inter-prediction stages; 2) assumes full computation/communication concurrency, which limits its applicability to only a certain type of devices; and 3) does not provide means for automatic management of collaborative execution. In [11], the automatic tuning of the number of RFs is performed only for a fixed (and smallest) SA size. Hence, we provide herein a completely different approach for inter-prediction adaptive scheduling and load balancing, when directly compared with [11]. Furthermore, the approach from [12] only considers collaborative *inter-prediction* based on geometrical data partitioning and functional performance modeling, i.e., it applies different load balancing strategy. To reduce the problem complexity, the approach from [12] replicates INT across devices and it does not consider any computation/communication overlapping.

This paper also advances our contributions from [13], by proposing a fully autonomous unified interloop video encoding framework for modern CPU + GPU systems. For the first time, this paper presents the communication minimization algorithm for heterogeneous parallel systems when two encoding modules share the access to a single data buffer. Furthermore, a complete set of possible framework configurations is presented according to the automatically identified amount of supported concurrency between the computation and communication at the accelerators. This paper also proposes a novel adaptive approach for automatic tuning of several video encoding parameters according to the current state of the platform and the real-time limit.

By relying on a row-based frame partitioning, the work proposed herein also tackles the efficient multiapplication divisible load theory (DLT) scheduling, which is not yet completely covered in the literature [14]. Moreover, there is only a limited number of studies targeting the DLT scheduling in CPU + GPU systems even for general problems [15].

III. Framework for Inter-Loop Video Encoding on Heterogenous Systems

The proposed unified framework aims at fully exploiting the capabilities of multicore CPU and multi-GPU platforms for collaborative interloop video encoding. This framework integrates several functional blocks, as shown in Fig. 2.

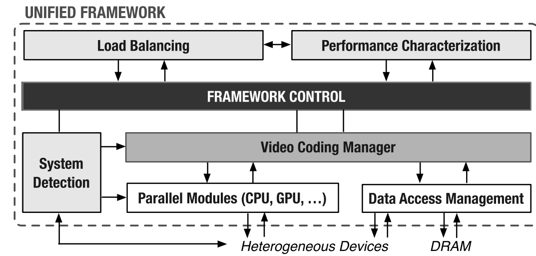

Fig. 2. Unified collaborative video encoding framework.

Algorithm 1 Framework Control (Main Functionality)

1: invoke *System Detection* to identify the number, type and characteristics of available computing 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 interconnection links for each accelerator
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**

The framework control block provides the key functionality by invoking all other functional blocks, i.e., *system detection*, *video coding manager*, *load balancing*, and *performance characterization*. The system detection block allows automatic identification of available devices and their capabilities. This information is further used to configure the functionality of the remaining blocks. Video coding manager orchestrates the collaborative encoding by invoking the respective implementations of parallel modules on the detected devices. It also invokes automatic data access management between the system main memory [dynamic RAM (DRAM)] and local memories at each accelerator. The collaborative execution is conducted by relying on the load distributions from the load balancing block, which is tightly coupled with an on-the-fly performance characterization of devices and communication links.

A. Framework Control and System Detection

As presented in Algorithm 1, *framework control* comprises two distinct execution phases, i.e., *initialization phase* (lines 1–6) and *iterative phase* (lines 7–11). The *initialization phase* is applied when encoding the first interframe, whereas the *iterative phase* is used for each subsequent interframe.

In the initialization phase, *framework control* first invokes the *system detection* block (line 1). This block identifies

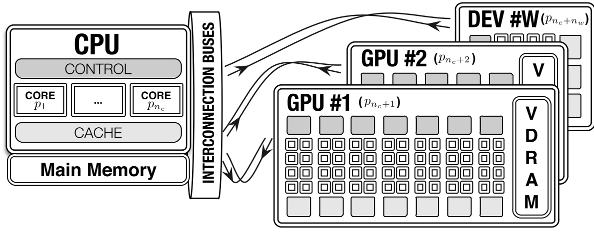

Fig. 3. Heterogeneous multicore CPU and multi-GPU/accelerator system.

all available *heterogeneous devices* in the platform and their architecture-specific capabilities. According to this information, the functionality of the remaining framework blocks is configured (line 2). For example, different implementations of *parallel modules* are instantiated based on the detected device microarchitectures and the supported type of single instruction multiple data (SIMD) vector intrinsics.

As presented in Fig. 3, heterogenous systems typically incorporate a set of n_c CPU cores and n_w accelerators, i.e., p_i devices, where $i = \{1, \ldots, n_c+n_w\}$. In such platforms, the accelerators are usually not stand-alone processing devices. 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. In detail, 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). For devices with dual copy engine, it is also possible to overlap the transfers in the opposite directions. Hence, the identification of these properties is crucial to instruct the configuration of *video coding manager* and *data access management* according to the detected system/device capabilities.

After these configurations, *framework control* invokes the *load balancing* routine (line 3 in Algorithm 1). When invoking *load balancing* for the first time, the overall workload of computationally intensive modules (i.e., ME, INT, and SME) is equidistantly partitioned among all devices. Since the proposed approach does not rely on any assumption about the performance of devices, interconnection links, or other system resources, these distributions are also used to obtain the initial *performance characterization* parameters. Hence, upon receiving the distributions, *framework control* initiates the on-device executions and automatic data transfers via *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 sent to *performance characterization* (line 6).

In the iterative phase, when encoding each subsequent interframe, *framework control* first invokes the *load balancing* routine (line 8). Based on a realistic *performance characterization* of system resources, 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. These distributions represent the number of MB rows to be processed on each p_i device, such that $\sum_{i=1}^{n_w+n_c} m_i = \sum_{i=1}^{n_w+n_c} l_i = \sum_{i=1}^{n_w+n_c} s_i = N$. By relying on the *video coding manager* and *data access management* configurations, all *parallel modules* are collaboratively executed (line 9). The recorded execution/transfer times are used to improve the accuracy of *performance characterization* (line 10).

In the proposed method, the load distribution is implemented at the level of individual MB rows, since it provides low scheduling overheads, while efficiently exploiting communication bandwidth and device performance. In contrast, at the finer-grained level (i.e., MBs), the latency might dominate the execution, and an inevitable repacking of the original frame format from a matrix/array of pixels (in raster scan order) to an array of structures (MBs) would be required.

To ensure an efficient control of the collaborative execution, the proposed framework was developed with vendor-specific programming models, i.e., OpenMP for CPUs and compute unified device architecture (CUDA) for GPUs. The different OpenMP threads control both CPU and GPU processing, where the GPU data transfers are performed in a completely asynchronous manner. Hence, the communication overheads are hidden by simultaneous CPU computations (independent of the transferred data). In different *GPU streams*,[1] this communication is also overlapped with the GPU computation, and with the communication in the opposite direction. To minimize transfer overheads, data allocations were performed in pinned memory regions.

IV. Video Coding and Data Access Management

Video coding manager orchestrates the collaborative interloop encoding in CPU + GPU systems by invoking *parallel modules* and automatic *data access management*. As previously referred, the *video coding manager* functionality is defined during the initialization phase. In addition to the detected device characteristics, the functionality of this block is also determined by the strict interloop data dependencies. In H.264/AVC (Section II), parallel execution at the entire interloop level brings to practice several hard-to-solve challenges. These challenges must be explicitly considered to ensure the correctness of the overall encoding from the aspect of sources of concurrency and inherent data dependencies.

An efficient parallelization must cope with data dependencies at several levels: 1) between consecutive frames; 2) within a single frame; and 3) between the encoding modules. In the H.264/AVC interloop, several frames cannot be encoded in parallel, since the CF encoding can only start after the previous frames are encoded and the required RFs are reconstructed. Moreover, the inherent data dependencies between the neighboring MBs in certain interloop modules (e.g., DBL) also limit the possibility to concurrently perform the entire encoding on different parts of a frame. Hence, efficient pipelined schemes with several modules can hardly be adopted, either for the parts of the frame or for the entire frame.

[1]GPU stream is a sequence of data transfers and kernel invocations issued by the CPU; different streams may concurrently execute their commands.

Furthermore, the output of one module is often the input for another (e.g., MVs from ME define the initial search point for SME), which imposes additional data dependencies between the interloop modules. Hence, all data-dependent interloop modules have to be sequentially processed (within a single frame). The only exceptions are ME and INT that can be simultaneously processed, since both use CF and/or RFs.

To efficiently exploit GPU capabilities, a common approach is to perform ME based on full-search block-matching (FSBM) [7]–[13]. Although the FSBM algorithm imposes high computational demands (i.e., all matching candidates need to be examined), it is suitable for data-level GPU parallelization and collaborative processing due to: 1) identical operations performed on each prediction candidate; 2) absence of branches; 3) regularity of memory accesses; and 4) independence on the video content [16]. In contrast, the adaptive ME algorithms do not have these properties, thus their GPU parallelization usually results in attaining poor performance. For example, the performance of UMHexagonS GPU implementation proposed in [17] does not surpass the one of multicore CPU. The adaptive algorithms also highly depend on the video content, thus it is hard to predict their performance.

To exploit the fine-grained GPU parallelism, a sufficient amount of data-independent computations is required. Hence, the spatial data dependencies need to be relaxed even in the FSBM algorithm, i.e., the dependencies imposed when relying on adjacent MVs to define the SA center. In contrast to the usually applied zero predictor [7], we rely on the temporal predictor defined as the best MV for collocated MB in the previous frame for the 16×16 partitioning mode [16]. The MVs are recalculated at the end of ME to provide the correct offsets to the median predictors. According to the rate–distortion (RD) analysis presented in Section VI, FSBM with this predictor provides the adequate coding efficiency, when compared with the original FSBM and UMHexagonS.

According to the performance of the CPU and GPU parallel algorithms and inherent data dependencies [4], the interloop modules are divided in two groups: 1) ME, SME and INT modules and 2) R* modules (i.e., MC, TQ, TQ^{-1} and DBL). Due to the low computational demands of MC, TQ, and TQ^{-1} (less than 3%) [4] and the limited possibility to collaboratively perform the DBL, the R* modules are assigned to a single (fastest) device [11]. In fact, increased scheduling and communication overheads when distributing the loads for modules with such a low complexity would hardly compensate for any performance gains. For brevity, the single-device mapping of the entire R* block is considered herein. If a single GPU (GPU_1) is assigned to perform all the R* modules, the derived procedure is designated as *GPU-centric*. In the *CPU-centric* approach, the R* modules are performed on all CPU cores.

Fig. 4 shows all possible configurations of *video coding manager* for accelerators with both single-copy and dual-copy engines. The presented configurations also include the orchestration of *parallel modules* and the inevitable data transfers. As it can be observed, *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 block instantiates the parallel algorithms and maps them to different devices. At the same time, specific input parameters (i.e., RF indexes) and required parts of the frame are supplied by *data access management*. In addition, *video coding manager* also provides the facilities to measure the execution and transfer times, thus allowing *performance characterization* of the system resources.

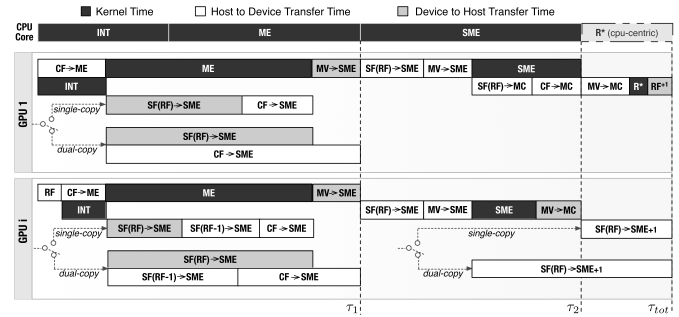

Fig. 4. Possible configurations of the video coding manager.

With respect to the previously referred interloop data-dependencies, several cross-device synchronization points are defined (see τ_1, τ_2, and τ_{tot} in Fig. 4). In brief, τ_1 reflects the dependency of SME on the outputs of ME and INT, while τ_2 marks the completion of SME and beginning of R* processing. The main objective of the proposed framework is to minimize the total interloop video encoding time, i.e., τ_{tot}. In detail, the main functionality of *video coding manager* is as follows.

1) τ_1 denotes the time when m_i and l_i portions of ME and INT are processed on each device p_i. In this period, the host to device transfers of the CF portions for ME (CF $\rightarrow$ ME) and for SME (CF $\rightarrow$ SME) are performed, as well as device to host transfers of the produced part of the interpolated SF (SF(RF) $\rightarrow$ SME). For accelerators that do not compute the R* modules (GPU_i), it is also required to fetch the resulting data from the previously encoded frame, namely: 1) the reconstructed RF before performing ME and INT (RF) and 2) the remaining portion of the previously interpolated SF, SF(RF $-$ 1) $\rightarrow$ SME, to complete SF at each accelerator. Upon finishing ME at each accelerator, the computed MVs are also transferred to the host (MV $\rightarrow$ SME). These input and output data transfers are sequentially performed for devices with single-copy engine. For accelerators with dual-copy engine, the device to host SF(RF) $\rightarrow$ SME and MV $\rightarrow$ SME transfers can occur in parallel with the host to device SF(RF $-$ 1) $\rightarrow$ SME and CF $\rightarrow$ SME transfers. The parallelism across multiple independent modules is also exploited, i.e., ME and INT.
2) τ_2 is the time when SME is finished on all devices (according to the s_i distributions). The missing parts of SF (SF(RF) $\rightarrow$ SME) and MVs (MV $\rightarrow$ SME) are transferred from the host, i.e., SF and MV parts previously computed in INT and ME on other devices. After τ_1, SME begins on all CPU cores, since all MVs

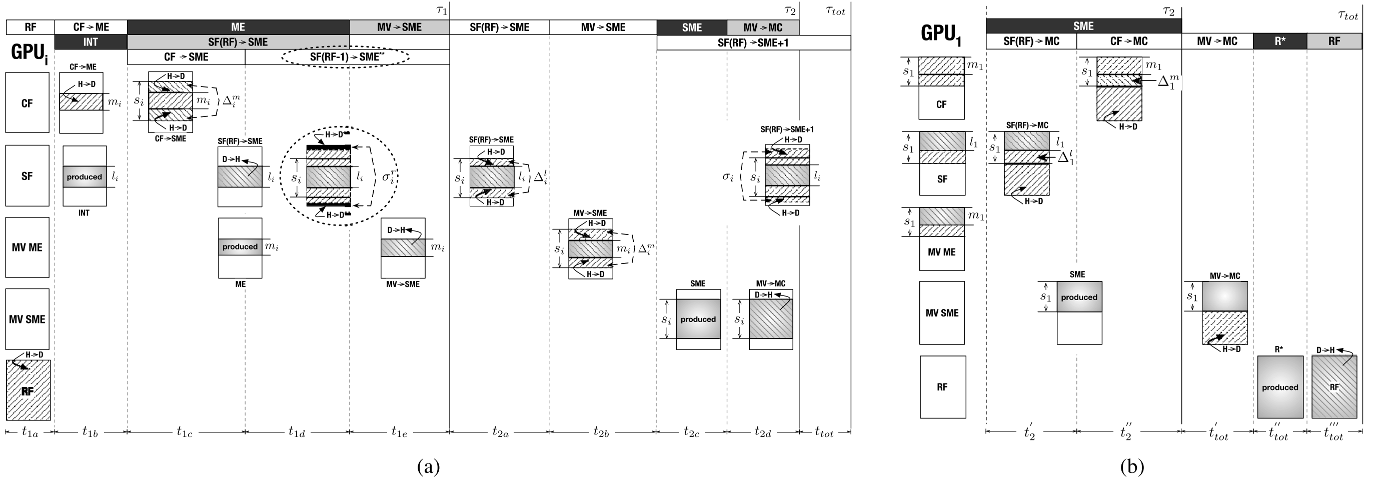

Fig. 5. State of input/output buffers for accelerators with dual-copy engine, when assigning m_i, l_i, and s_i loads to (a) GPU_i and (b) GPU_1.

are already present. For the accelerator that computes the R* modules (GPU_1), in parallel with SME, the remaining parts of SF (SF $\rightarrow$ MC) and CF (CF $\rightarrow$ MC) are transferred from the host (to perform MC after τ_2). For the other accelerators (GPU_i), after the computation of the SME portion, the computed MVs are transferred back to the host (MV $\rightarrow$ MC).

3) τ_{tot} denotes the overall interloop encoding time for a single interframe. In GPU-centric approach, prior to computation of the R* modules on GPU_1, the missing MVs from SME are transferred from the host (MV $\rightarrow$ MC), i.e., MVs computed on other devices. After the R* modules are processed on GPU_1, the reconstructed RF (RF^{+1}) is sent back to the host to allow its transfer at the beginning of the next interframe encoding on the other accelerators. At the same time, for the GPU_i accelerators with single-copy engine, the host to device transfers of the remaining SF part (SF $\rightarrow$ SME + 1) are performed. In the case of GPU_i accelerators with dual-copy engine, this transfer can be initiated at the time when SME is started.

Since the proposed solution strictly respects the data dependencies at different interloop levels, as specified in H.264/AVC, the resulting bitstream is fully compliant with this standard. Furthermore, some of the addressed challenges and proposed solutions can also be applied for other video coding standards. For example, HEVC/H.265 [3] mostly preserves the functionality of processing modules and inherent data-dependencies from the H.264/AVC standard. In fact, HEVC/H.265 allows even more efficient data-level parallelization due to the higher computational demands of ME, SME, and INT, and increased parallelization potential of DBL.

A. Parallel Modules

During the orchestration of the encoding procedure, *video coding manager* performs mapping of the instantiated *parallel modules* to the detected devices. To provide a wide support for different device architectures, an extensive library of highly optimized interloop modules was specifically developed[2] with vendor-specific programming models and tools (i.e., OpenMP and SIMD intrinsics for CPUs, and CUDA/parallel thread execution for GPUs). Parallel CPU algorithms exploit both coarse-grained (between the cores) and fine-grained parallelism (by applying SIMD instructions on successive memory locations). For different CPU microarchitectures, i.e., Intel Nehalem, Sandy/Ivy Bridge, and Haswell, optimized implementations for each module are provided, based on streaming SIMD extensions (SSE) 4.2, advanced vector extensions (AVX), and AVX2 intrinsics. For different GPU architectures, i.e., Nvidia Tesla, Fermi, and Kepler, parallel algorithms are developed for each module. These algorithms exploit the fine-grained data parallelism and efficiently use the GPU memory hierarchy. As a result, the adopted parallelization approaches vary depending on the device architectures, computational load of the encoding modules, inherent data dependencies, and locality and regularity of data access patterns. Considering the limited space in this paper, the detailed description of the developed parallel algorithms is provided in [11] and [16].

B. Data Access Management

The *data access management* block includes specific mechanisms for device memory management and automatic data transfers in CPU + GPU systems. Fig. 5 shows the main functionality of this block. For each synchronization point from Fig. 4, the state of the input and output buffers (i.e., CF, SF, RF, and MVs from ME and SME) are presented for different accelerator roles (i.e., GPU_i and GPU_1) and for different parts of the encoding procedure. In the proposed solution, each device has its own local copy of the input/output buffers. To minimize the communication volume, only the portions of these buffers are transferred to/from the devices according to the distributions from the *load balancing* routine and the *video coding manager* functionality (Fig. 4).

As presented in Fig. 5(a), for the accelerators that do not compute the R* modules (GPU_i), the entire previously

[2]The library is available upon request to the corresponding authors.

reconstructed RF is first received in the t_{1a} interval. In the t_{1b} interval, the CF portion is transferred from the host (CF $\rightarrow$ ME) according to the m_i distribution, i.e., the number of input MB rows from the CF to be processed in the ME. The MB row size in the CF (r_{CF}) is calculated as $r_{\mathrm{CF}} = h_{\mathrm{MB}} \times w_{\mathrm{CF}}$, where h_{MB} is the MB height and w_{CF} is the CF width (in pixels). Hence, the amount of transferred data to the GPU$_i$, in the t_{1b} interval, is equal to $m_i \times r_{\mathrm{CF}}$. However, these data must also refer to the adequate position in the CF, such that each device performs the ME on separate portion of the CF in respect of the assigned m_i distribution. For the GPU$_i$, the starting position in the CF is calculated as $\sum_{j=1}^{i-1} m_j \times r_{\mathrm{CF}}$, where m_j are the ME distributions for all previously enumerated devices ($j < i$).

In the t_{1b} interval, INT is also started on the GPU$_i$ according to the l_i distribution. This distribution refers to the l_i MB rows from the RF, such that the l_i MB rows in the interpolated SF are *produced*. The MB row in the SF (r_{SF}) is 16 times larger than the MB row from the RF ($r_{\mathrm{RF}} = r_{\mathrm{CF}}$), since each RF pixel corresponds to 16 interpolated SF pixels, i.e., 1 full-, 3 half- and 12 quarter-pixels. The *produced* portion of the SF at the GPU$_i$ must be stored at the adequate starting position in the SF calculated as $\sum_{j=1}^{i-1} l_j \times r_{\mathrm{SF}}$, where l_j are the INT distributions for all previously enumerated devices ($j < i$).

In the t_{1c} and t_{1d} periods, ME is performed on the previously received m_i MB rows from the CF (see t_{1b}). In the ME, m_i MB rows from the MV are *produced* on the GPU$_i$. The MB row size in the MV_ME buffer (r_{MV}) is equal to the number of MBs in a row ($w_{\mathrm{CF}}/w_{\mathrm{MB}}$) multiplied by the size of all MVs produced for a single MB [16]. Hence, the total amount of *produced* MVs in the ME is equal to $m_i \times r_{\mathrm{MV}}$. The *produced* MVs in the MV_ME are placed at the starting position calculated as $\sum_{j=1}^{i-1} m_j \times r_{\mathrm{MV}}$, where m_j are the ME distributions for all previously enumerated devices ($j < i$). In t_{1c} and t_{1d}, the SF portion interpolated on the GPU$_i$ ($l_i \times r_{\mathrm{SF}}$) is also transferred to the host, SF(RF) $\rightarrow$ SME. In t_{1e}, the computed MVs from ME are sent to the host (MV $\rightarrow$ SME).

For GPU$_i$ accelerators with dual-copy engine, after the t_{1b} interval [see t_{1c} in Fig. 5(a)], the additional CF portions required for the SME are transferred from the host (CF $\rightarrow$ SME). The amount and the number of these additional transfers depend on the s_i and m_i distributions for the GPU$_i$, and on their starting positions in the CF. As before, these positions are computed from the s_j and m_j distributions for all previously enumerated devices ($j < i$). In the case shown in Fig. 5(a), two *additional transfers* (Δ_i^m) are performed to fetch the upper part and the bottom part of the CF portion required for SME.

After the additional CF transfers, the remaining part of the previously interpolated SF (during the encoding of previous interframe) is transferred to the GPU$_i$ in the t_{1d} and t_{1e} periods (SF(RF $-$ 1) $\rightarrow$ SME). In detail, to ensure the correctness of ME and SME for the subsequent interframes, all devices need to have the complete lists of previously reconstructed RFs and interpolated SFs. Since the RF is reconstructed only at the end of the interloop (after processing R* modules), the most recently reconstructed RF is fetched in the t_{1a} period and locally stored in a circular buffer at the GPU$_i$. However, collaboratively interpolated SF is available after INT and the device to host SF(RF) $\rightarrow$ SME transfers, i.e., after t_{1d}.

Since a large volume of data needs to be transferred ($r_{\mathrm{SF}} = 16 \times r_{\mathrm{RF}}$), the *remaining parts* of the SF are fetched in two steps, such that the SF is completed on each accelerator as soon as possible. In the first step, the remaining transfers are performed during the encoding of the current interframe, i.e., SF(RF) $\rightarrow$ SME$+1$ in t_{2c}, t_{2d}, and t_{tot}. In the second step, the *remaining transfers* are performed during the encoding of the next interframe, i.e., SF(RF $-$ 1) $\rightarrow$ SME in t_{1d} and t_{1e}. The amount and the number of *remaining transfers* in each step depend on the s_i and l_i distributions for the GPU$_i$, on their starting positions in the SF and on the amount of *additional transfers* (Δ_i^l). In the first step, the amount of *remaining transfers* (σ_i) is determined as the maximum number of MB rows from the SF that can be fetched in the t_{2c}, t_{2d}, and t_{tot} periods (Fig. 4). In the second step, the amount of *remaining transfers* (σ_i^r) is determined as the number of MB rows that are still missing to complete the SF at the accelerator.

To ease the explanation of this procedure, the remaining transfers in the second step are presented in Fig. 5(a) as the SF remainder that is fetched in the next iteration (see the dashed circled SF buffer, where the remaining portions are marked with dark solid colors). The remaining transfers in the first step are marked as SF(RF) $\rightarrow$ SME$+1$ in t_{2c}, t_{2d}, and t_{tot}.

The proposed framework integrates a specific procedure to determine the number and the amount of *additional and remaining transfers* when the load distributions for different modules refer to the data located in the same buffer. This procedure maximizes the reuse of data already available on devices and reduces the overall communication volume. It considers the relative distance between the distributions from different modules for the same device as well as the distributions from the previously enumerated devices. This procedure is explained in detail in the Appendix and it is used to determine the amount of additional Δ_i^m and Δ_i^l transfers and the amount of remaining σ_i and σ_i^r transfers in Fig. 5(a).

In the t_{2a} interval, the additional part of the SF for SME is received, i.e., SF(RF) $\rightarrow$ SME represented as two data transfers in Fig. 5(a). The amount of these additional transfers (Δ_i^l) is computed by applying the above-referred procedure on the s_i and l_i distributions for the GPU$_i$ and their starting positions in the SF. As before, the starting positions are computed from the s_j and l_j distributions for all previously enumerated devices ($j < i$). Similarly, the additional MVs required for SME (Δ_i^m) are received in t_{2b}, i.e., two data transfers for MV $\rightarrow$ SME.

In the t_{2c} period, SME is performed on the previously received s_i MB rows from the CF, MV_ME, and SF buffers. In the SME on the GPU$_i$, the total amount of $s_i \times r_{\mathrm{MV}}$ MB rows from MV_SME is *produced*. The *produced* MVs in MV_SME are placed at the starting position calculated as $\sum_{j=1}^{i-1} s_j \times r_{\mathrm{MV}}$, where s_j are the SME distributions for all previously enumerated devices ($j < i$). In t_{2d}, the computed MVs from the SME are transferred to the host (MV $\rightarrow$ MC).

In the initial stages, the overall state of the input and output buffers for the accelerator that performs the R* modules (GPU_1) does not significantly differ from the other accelerators (GPU_i). However, as shown in Fig. 5(b), while the MVs from SME are *produced* on the GPU_1 in t'_2 and t''_2, several *remaining transfers* are performed from the host. According to the amount of the remaining transfers (computed by applying the procedure from the Appendix), the remaining SF parts are first fetched, i.e., SF(RF) $\rightarrow$ MC in t'_2. Then the remaining parts of the CF (CF $\rightarrow$ MC) are transferred in t''_2. In the t'_{tot} period, the remaining part of the SME MVs is also transferred from the host to the GPU_1 (MV $\rightarrow$ MC). Then the R* modules are computed in the t''_{tot} interval. Upon the RF is reconstructed, it is transferred to the host in the t'''_{tot} period.

During these time periods, the interloop modules are also performed on all CPU cores in parallel. As presented in Fig. 4, INT and ME are computed within the τ_1 interval, while SME is completed between τ_1 and τ_2. In the CPU-centric configuration, the R* modules are computed on the CPU cores in the interval between τ_2 and τ_{tot}.

V. Load Balancing and Performance Characterization

To cope with the inherent data dependencies and the overall complexity of the interloop encoding, a specific *load balancing* routine is proposed herein for heterogeneous CPU + GPU systems. This routine aims at efficiently exploiting the computation power of devices, the asymmetric bandwidth of communication links and the overlapping between the computation and communication at the accelerators.

The proposed *load balancing* routine differs according to: 1) the type of device selected to perform the remaining R* modules (CPU-centric or GPU-centric approach) and 2) the capabilities of the accelerators to support different amounts of concurrency between data transfers and computations (single-copy or dual-copy engines). Algorithm 2 presents all possible variants of the *load balancing* routine for CPU + GPU systems, i.e., different configurations of *video coding manager* and *data access management* (Figs. 4 and 5).

Algorithm 2 Load Balancing for CPU + GPU Video Encoding

Input: $N, n_w, n_c, T_1^{R*}, T_c^{R*}, K_i^m, K_i^l, K_i^s$

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 τ_{tot}

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

$\forall i \in \{n_w+1, \ldots, n_w+n_c\}$: /** *CPU code* **/

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

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

IF CPU-centric **THEN**

$$| \ \tau_2 + T_c^{R*} \leq \tau_{\text{tot}} \quad (4)$$

ENDIF

IF GPU-centric **THEN** /** *GPU_1 code* **/

$$| \ m_1 K_1^{cfhd} + m_1 K_1^m + m_1 K_1^{mvdh} \leq \tau_1 \quad (5)$$

| **IF** single-copy engine **THEN**

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

$$| \ | \ m_1 K_1^{cfhd} + l_1 K_1^{sfdh} + \Delta_1^m K_1^{cfhd} + m_1 K_1^{mvdh} \leq \tau_1 \quad (7)$$

| **ELSE IF** dual-copy engine **THEN**

$$| \ | \ l_1 K_1^l + l_1 K_1^{sfdh} + m_1 K_1^{mvdh} \leq \tau_1 \quad (8)$$

$$| \ | \ l_1 K_1^l + \Delta_1^m K_1^{cfhd} \leq \tau_1 \quad (9)$$

$$| \ | \ m_1 K_1^{cfhd} + \Delta_1^m K_1^{cfhd} \leq \tau_1 \quad (10)$$

| **ENDIF**

$$| \ \tau_1 + \Delta_1^l K_1^{sfhd} + \Delta_1^m K_1^{mvhd} + s_1 K_1^s \leq \tau_2 \quad (11)$$

$$| \ \tau_1 + \Delta_1^l K_1^{sfhd} + \Delta_1^m K_1^{mvhd} + (N - l_1 - \Delta_1^l) K_1^{sfhd} + (N - m_1 - \Delta_1^m) K_1^{cfhd} \leq \tau_2 \quad (12)$$

$$| \ \tau_2 + (N - s_1) K_1^{mvhd} + T_1^{R*} + N K_1^{rfdh} \leq \tau_{\text{tot}} \quad (13)$$

ENDIF

$\forall i \in \{2, \ldots, n_w\}$: /** *GPU_i code* **/

$$N K_i^{rfhd} + m_i K_i^{cfhd} + m_i K_i^m + m_i K_i^{mvdh} \leq \tau_1 \quad (14)$$

IF single-copy engine **THEN**

$$| \ N K_i^{rfhd} + l_i K_i^l + l_i K_i^{sfdh} + \sigma_i^{r-1} K_i^{sfhd} + \Delta_i^m K_i^{cfhd} + m_i K_i^{mvdh} \leq \tau_1 \quad (15)$$

$$| \ N K_i^{rfhd} + m_i K_i^{cfhd} + l_i K_i^{sfdh} + \sigma_i^{r-1} K_i^{sfhd} + \Delta_i^m K_i^{cfhd} + m_i K_i^{mvdh} \leq \tau_1 \quad (16)$$

ELSE IF dual-copy engine **THEN**

$$| \ N K_i^{rfhd} + l_i K_i^l + l_i K_i^{sfdh} + m_i K_i^{mvdh} \leq \tau_1 \quad (17)$$

$$| \ N K_i^{rfhd} + l_i K_i^l + \sigma_i^{r-1} K_i^{sfhd} + \Delta_i^m K_i^{cfhd} \leq \tau_1 \quad (18)$$

$$| \ N K_i^{rfhd} + m_i K_i^{cfhd} + \sigma_i^{r-1} K_i^{sfhd} + \Delta_i^m K_i^{cfhd} \leq \tau_1 \quad (19)$$

ENDIF

$$\tau_1 + \Delta_i^l K_i^{sfhd} + \Delta_i^m K_i^{mvhd} + s_i K_i^s + s_i K_i^{mvdh} \leq \tau_2 \quad (20)$$

IF single-copy engine **THEN** /** *the remaining transfers (see Appendix VII)* **/

$$| \ \sigma_i = \text{MIN}(N - l_i - \Delta_i^l, (\tau_{\text{tot}} - \tau_2)/K_i^{sfhd}) \quad (21)$$

ELSE IF dual-copy engine **THEN**

$$| \ \sigma_i = \text{MIN}(N - l_i - \Delta_i^l, (\tau_{\text{tot}} - \tau_1 - \Delta_i^l K_i^{sfhd} - \Delta_i^m K_i^{mvhd})/K_i^{sfhd}) \quad (22)$$

ENDIF

$$\sigma_i^r = N - l_i - \Delta_i^l - \sigma_i \quad (23)$$

$\forall i \in \{1, \ldots, n_w\}$: /** *the additional transfers (see Appendix VII)* **/

$$\Delta_i^m = \text{MS_BOUNDS}(m, s) \quad (24)$$

$$\Delta_i^l = \text{LS_BOUNDS}(l, s) \quad (25)$$

The *load balancing* routine is based on linear programming [18] and it perceives the collaborative interloop encoding as a unified optimization problem. It relies on *performance characterization* to distribute the loads of the ME, INT, and SME modules and to map the R* modules to the fastest device. To minimize the total encoding time (τ_{tot}), several distribution vectors are determined, 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 p_i device. According to (1) from Algorithm 2, the sum of distributions for each module must be equal to the total number of MB rows (N).

As previously referred, the currently interpolated SF is completed at each accelerator in two steps, i.e., with two *remaining transfers*. Hence, two distribution vectors are determined for each accelerator, i.e., $\sigma = \{\sigma_i\}$ and $\sigma^r = \{\sigma_i^r\}$, $i = \{1, \ldots, n_w\}$. These distributions reflect the amount of *remaining transfers* (in MB rows) in each step. The $\sigma = \{\sigma_i\}$ vector represents the amount of remaining transfers in the first step, i.e., during the processing of the current interframe [see SF(RF) $\rightarrow$ SME + 1 in Fig. 5(a)]. The $\sigma^r = \{\sigma_i^r\}$ distributions represent the remaining transfers in the second step, i.e., to complete the SF at each p_i accelerator during the processing of the next interframe [see SF(RF − 1) $\rightarrow$ SME in Fig. 5(a)]. In fact, the $\sigma^r = \{\sigma_i^r\}$ remainders for the CF serve as inputs when processing the next interframe, which are designated as $\sigma^{r-1} = \{\sigma_i^{r-1}\}$.

The performance of each p_i device is characterized with K_i^m, K_i^l, and K_i^s parameters for the ME, INT, and SME modules, respectively. These parameters are expressed in processing time per MB row, obtained for the currently

TABLE I
PROCESSING DEVICES AND CONSIDERED HETEROGENEOUS CPU + GPU SYSTEMS

Systems	SysSS		SysDD			SysSD		
Configuration: *GPU-centric*	CPU_N + 2×GPU_F (n/a + single-copy + single-copy)		CPU_S + GPU_K$_2$ + GPU_K$_4$ (n/a + dual-copy + dual-copy)			CPU_H + GPU_K + GPU_K$_4$ (n/a + single-copy + dual-copy)		
Devices	CPU_N	GPU_F	CPU_S	GPU_K$_2$	GPU_K$_4$	CPU_H	GPU_K	GPU_K$_4$
Architecture	Intel Nehalem	NVIDIA Fermi	Intel Sandy Bridge	NVIDIA Kepler		Intel Haswell	NVIDIA Kepler	
Model	Core i7 950	GeForce 580GTX	Core i7 2600K	Tesla K20	Tesla K40	Core i7 4770K	GeForce 780GTX	Tesla K40
SIMD support	SSE4.2	yes	AVX	yes	SIMD-word	AVX2	SIMD-word	SIMD-word
# Copy engines	n/a	single-copy	n/a	dual-copy	dual-copy	n/a	single-copy	dual-copy
# Cores	4	512	4	2496	2880	4	2880	2880
Frequency	3 GHz	1.54 GHz	3.4 GHz	0.71 GHz	0.88 GHz	3.5 GHz	1.54 GHz	0.88 GHz
Memory	4 GB	1.5 GB	8 GB	5 GB	11.5 GB	32 GB	3 GB	11.5 GB

processed loads in m, l, and s vectors. The obtained parameters serve as inputs to the *load balancing* routine when determining the next distributions in the iterative phase of Algorithm 1. Similarly, 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 performs the R* modules, T_1^{R*} and T_c^{R*} refer to the time required to perform the complete R* sequence on the selected device, i.e., GPU$_1$ (GPU-centric) or CPU cores (CPU-centric), respectively. The *performance characterization* is updated in runtime (after each encoded frame), thus the proposed framework is capable of adapting the *load balancing* decisions according to the current state of the platform. This is important for video coding in unreliable and nondedicated systems, where the system performance can vary during the runtime.

According to the previously presented analyses, the minimization of the total interloop encoding time τ_{tot} is attained by satisfying different conditions from Algorithm 2. Due to the limited space in this paper, the overall *load balancing* functionality can be briefly summarized as follows.

1) Equations (2) and (3) express the conditions for multicore CPU execution in respect of τ_1 and τ_2 synchronization points from Fig. 4. For CPU-centric configuration, the condition (4) reflects R* processing until the τ_{tot} point.
2) Equations (5)–(13) present the conditions for the GPU-centric case, i.e., for GPU$_1$ performing the R* modules. Depending on the accelerator capabilities, they guarantee that all input/output transfers and kernel executions are performed according to the synchronization points in Figs. 4 and 5.
3) Equations (14)–(23) state the necessary conditions for the other employed accelerators (GPU$_i$) according to their capabilities and the execution scenarios presented in Figs. 4 and 5.
4) Equations (24) and (25) correspond to the previously referred procedures from the Appendix. These procedures determine the amount of additional transfers when two different modules share the access to a single data buffer. The MS_BOUNDS procedure determines the additional Δ_i^m transfers for ME and SME accessing the CF and MV buffers. The LS_BOUNDS procedure determines the additional Δ_i^l transfers for INT and SME accessing the SF.

VI. EXPERIMENTAL RESULTS

To challenge the real-time encoding of HD video sequences, the efficiency of the proposed framework was experimentally evaluated on several CPU + GPU systems. The systems equipped with devices of different capabilities and architectures were selected, as presented in Table I. In detail, the SysSS system is equipped with a quad-core Intel Nehalem CPU (CPU_N) and two Nvidia Fermi GPUs (GPU_F) with single-copy engines. The SysDD system relies on the quad-core Intel Sandy Bridge CPU and two Nvidia Tesla devices (K20 and K40), both with dual-copy engines. Finally, the SysSD system is based on the newest Intel Haswell quad-core CPU (CPU_H) and it combines the GPU devices with different capabilities, i.e., an Nvidia Kepler 780GTX (GPU_K) with single-copy engine and a Tesla K40 with dual-copy engine. The tests were performed on OpenSUSE 13.1 operating system with CUDA 5.5, Intel Parallel Studio 12.1, and OpenMP 3.0.

The proposed framework relies on the compression techniques from the H.264/AVC JM 18.4 encoder [19]. The encoding parameters specified by the JM configuration are used as the framework inputs. The experimental results in different systems were obtained by relying on the highly optimized *parallel modules* [11], [16] instantiated according to the characteristics of the CPU and GPU devices (e.g., SIMD support from Table I). The *parallel modules* follow the compression methods proposed in the JM software, and preserve the hard data dependencies specified by the H.264/AVC standard. The outputs of the modules are provided in the format used in the reminder of JM software, and they are compatible with the modules not instantiated from the framework library.

As referred to in Section IV, a fine grained data-level parallelization of the FSBM ME is achieved by relying on a temporary dependent SA-center predictor defined as the best MV for collocated MB in the previous frame for the 16 × 16 partitioning mode [16]. To evaluate the coding efficiency of the proposed solution, an extensive RD analysis was performed by strictly following the Video Coding Experts Group recommendations [20] for $IPPP$ sequences with four RFs, baseline profile, and the following quantization parameters (QPs) {ISlice, PSlice}: {22, 23}, {27, 28}, {32, 33}, and {37, 38}.[3] Fig. 6 presents the average peak SNR (PSNR) gains (in decibels) of the proposed solution (*Prop.*) and the

[3]The extensive RD analysis can be found in [16], where several candidates for the SA center predictors and different video resolutions were evaluated.

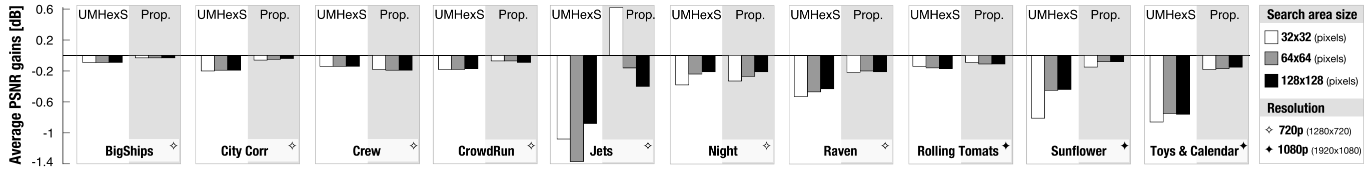

Fig. 6. Average PSNR gains of the proposed solution and UMHexagonS over the original FSBM algorithm for different SA sizes and HD video sequences.

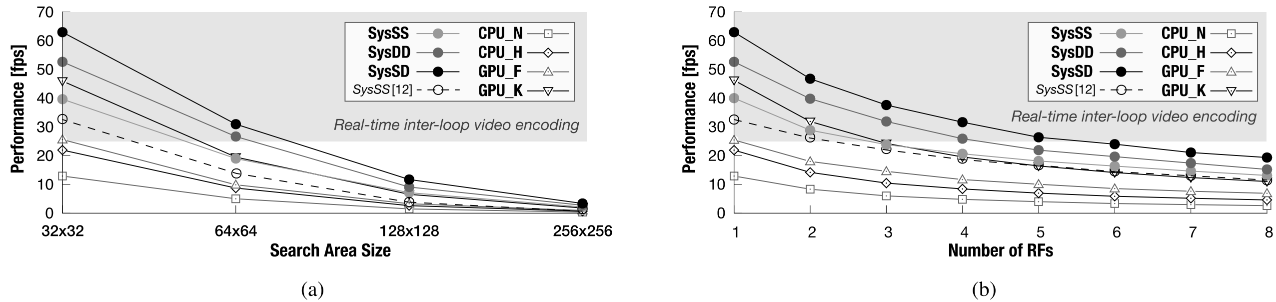

Fig. 7. Performance on different devices/systems and 1080p sequences for (a) different SA sizes (1 RF) and (b) different numbers of RFs (32 × 32 SA).

UMHexagonS algorithm (UMHexS) for different SA sizes and HD video sequences. The PSNR gains were obtained by calculating the Bjøndegaard objective difference [20] between the RD-curves of the tested algorithms and the RD-curve of the original FSBM in JM software.

As it can be observed in Fig. 6, the FSBM encoding with the *proposed* predictor outperforms the UMHexagonS in 86.7% tested cases, i.e., for different sequences and SAs. In fact, for Jets 720p sequence and 32 × 32 SA, even the original FSBM was outperformed. This phenomenon occurs due to the ability of the proposed temporal predictor to provide accurate motion prediction in the presence of a pan-left camera movement in the low-motion part of the Jets 720p video sequence. For larger SAs, both the original FSBM and the one based on the proposed predictor provide the adequate coding efficiency.

To further evaluate the efficiency of the proposed framework, the *load balancing* routine was applied to encode different 1080p sequences (*Toys and Calendar* and *Rolling Tomatoes*) with QP of {27, 28} for {ISlice, PSlice}. The performance was expressed in terms of the resulting encoding speed, i.e., time per frame or frames per second. In the presented charts, the shaded area marks the region where it is possible to achieve a real-time interloop encoding.

Fig. 7 presents the experimentally obtained performance with the proposed framework in the considered systems, for four different SA sizes and different number of RFs. In addition, the performance obtained on individual devices is also presented. For brevity, only the subset of the obtained results is included, i.e., for devices with the lowest (CPU_N and GPU_F) and the highest (CPU_H and GPU_K) performance capabilities. For comparison purposes, Fig. 7 also presents an additional performance curve, i.e., SysSS [11]. This curve represents the performance obtained with the approach from [11] in the SysSS platform for different SA sizes and different number of RFs. The SysSS platform was chosen, since the best performance was reported in [11] for this system.

Fig. 7(a) shows the scalability of the proposed approach over the SA size. As expected, the overall performance decreases between two successive SA sizes due to the quadruplication of the ME computational load. As it can be observed, higher performance was achieved for newer device architectures by instantiating the respective *parallel modules*. For example, the interloop encoding on CPU_H is about 1.7× faster than on CPU_N, while GPU_K outperforms GPU_F for almost 2×. On both GPUs, the real-time interloop encoding (≥25 frames/s) was achieved for the 32×32 SA size and one RF.

For tested CPU + GPU systems and all SA sizes in Fig. 7(a), the corresponding single device executions were significantly outperformed with the proposed framework. As a result, a real-time interloop encoding of 1080p sequences was achieved on all tested systems for the SA size of 32 × 32 and one RF. For SysSD and SysDD, a real-time encoding was achieved even for higher and more challenging 64 × 64 SA. In the SysSS platform, the proposed framework outperforms the approach from [11] for all tested cases, by providing on average 43.9% higher performance and a maximum of 61.2% for 32 × 32 SA size.

Fig. 7(b) presents the performance obtained with the proposed framework when encoding 1080p sequences for different number of RFs and SA size of 32×32. The real-time interloop encoding was achieved on all tested CPU + GPU systems for multiple RFs. In particular, it was achieved for up to five RFs on SysSD, outperforming both SysDD (up to four RFs) and SysSS (up to two RFs). Furthermore, in the SysSD, an average speedup of about 1.6 was obtained when compared with the single GPU_K, and about 3.7 when compared with CPU_H. In the SysSS, speedups of up to 2.2 and 5 were obtained, when compared with the GPU_F and quad-core CPU_N, respectively.

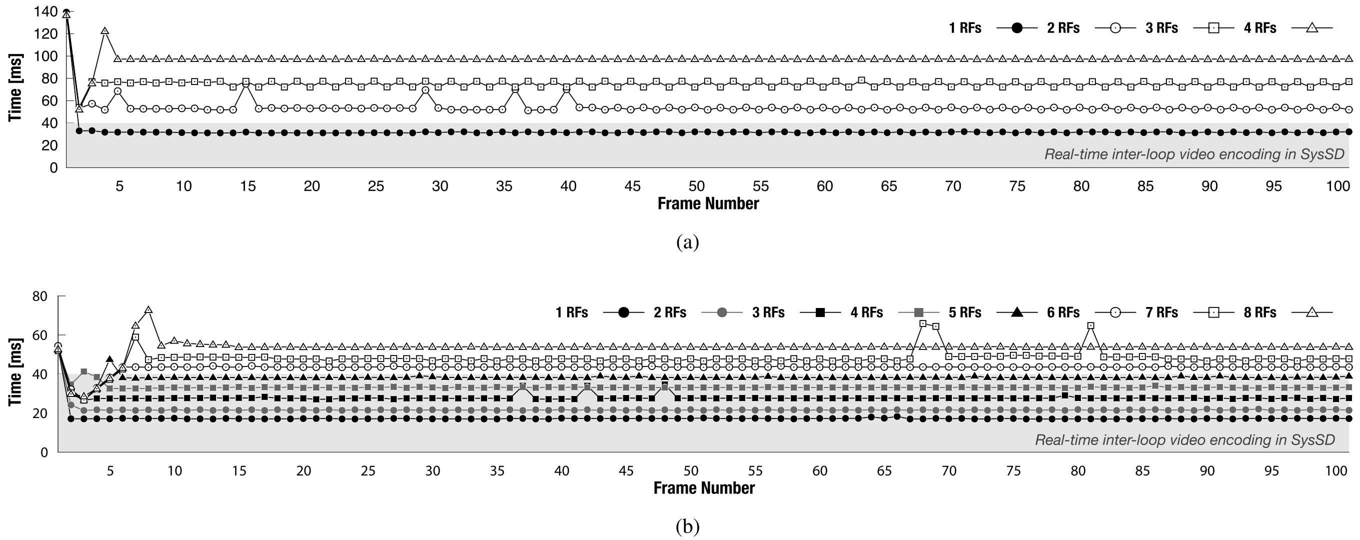

Fig. 8. Performance of the proposed approach for *Toys* and *Calendar* sequence (first 100 frames) with the SA sizes of (a) 64×64 and (b) 32×32 pixels.

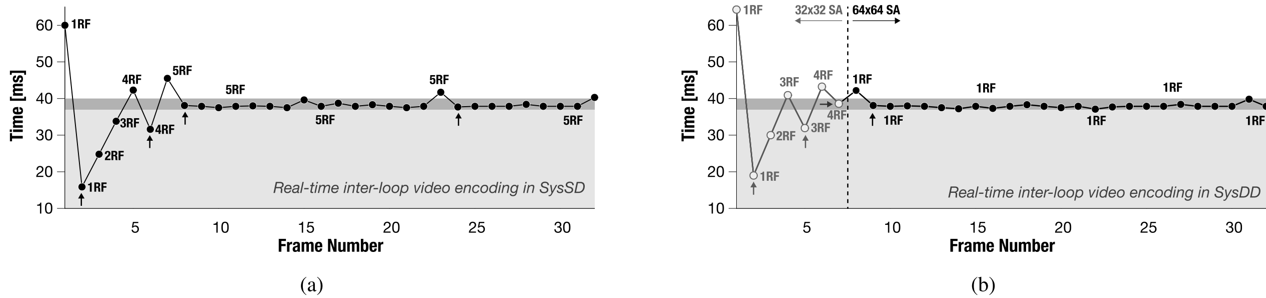

Fig. 9. Performance of the proposed automatic tuning procedure for 1080p sequences when tuning the (a) number of RFs (32×32 SA) in SysSD and (b) number of RFs and the SA size in the SysDD platform.

In the same platform, the proposed framework delivers a higher performance than the approach from [11] (providing a maximum gains of 19.5%), while outperforming the serial JM execution by several orders of magnitude.

The capabilities of the proposed framework to adapt the load balancing decisions according to the current state of the SysSD platform were experimentally verified when processing the first 100 interframes of a 1080p *Toys* and *Calendar* sequence. Fig. 8 shows the obtained encoding time for different numbers of RFs and SA sizes, namely: the SA of 64×64 in Fig. 8(a) and the SA of 32×32 in Fig. 8(b). In both cases, the time obtained when encoding the first interframe corresponds to an equidistant partitioning applied in the initialization phase of the *framework control* (Algorithm 1). After the initial *performance characterization*, the proposed *load balancing* was invoked for each subsequent interframe, i.e., the iterative phase of the *framework control* (Algorithm 2). Hence, a significant reduction in the encoding time was achieved, starting already with frame 2. As a result, a real-time encoding was achieved for different numbers of RFs, i.e., one RF for 64×64 SA and five RFs for 32×32 SA.

For a higher number of RFs [e.g., for six RFs in Fig. 8(b)], the encoding time increases with the number of encoded interframes at the beginning (frames 2–5), after which it becomes near-constant (frame 6). This effect occurs because a single RF is produced during the encoding of a single interframe (Fig. 5). Hence, the number of considered RFs increments with each processed interframe, until reaching the specified number of RFs. Therefore, Fig. 8 shows the ability of the proposed *load balancing* to cope with this increasing complexity, by efficiently distributing the loads.

An interesting phenomenon was observed during the encoding with three RFs (frames 37, 42, and 48) and seven RFs (frames 68 and 81) in Fig. 8(b), and with two RFs (frames 5, 15, 29, 36 and 40) in Fig. 8(a), where sudden changes in the system performance occurred. However, the dynamic *performance characterization* allowed capturing these unexpected changes, thus resulting in a successful load redistribution according to the new state of the platform. This can be evidenced in a very fast recovery of the performance curves, which required at most two interframes to converge to the regions with stable load distributions. Finally, the overheads introduced by the proposed framework take, on average, less than 2 ms per interframe encoding, which is significantly less than the time required to execute any individual interloop module.

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

The proposed adaptive *load balancing* allows on-the-fly tuning of the video coding parameters according to the target encoding performance. For example, some encoding parameters, such as the number of RFs and/or SA size, can be adaptively adjusted to achieve the predefined goals.

In addition, the proposed adaptive procedure allows minimization of the scheduling overhead, by reducing the number of algorithm invocations. Hence, the load balancing distributions determined for previous frames can be reused when encoding the next interframes. These distributions can also be reused when encoding with the increased number of RFs or SA size, as long as the real-time processing is achieved. Once the real-time encoding is not achieved, the *performance characterization* is updated and the *load balancing* is invoked (see Algorithm 2).

Fig. 9 presents the obtained performance with the proposed adaptive procedures in different CPU + GPU platforms, when automatically tuning the number of RFs [Fig. 9(a)], and both the number of RFs and the SA size [Fig. 9(b)]. These procedures determine the upper bounds on the targeted encoding parameters, such that a real-time encoding is achieved ($\approx$40 ms per frame). To cope with performance variations in nondedicated systems, the threshold values were defined to designate the region of realistically attainable real-time processing (see the dark gray regions in Fig. 9).

Fig. 9(a) shows the achieved performance with the proposed adaptive algorithm in the SysSD, when automatically tuning the number of RFs for the SA size of 32×32. When encoding the first interframe, the equidistant partitioning was applied. Since the real-time encoding was not achieved, the *performance characterization* was updated. Then the *load balancing* was invoked to determine the distributions for the encoding of the second frame with one RF [see the arrow in Fig. 9(a)]. In the proposed adaptive algorithm, the determined distributions are reused when encoding the subsequent interframes, if the real-time encoding is achieved. In addition, if the achieved performance is also below the threshold limit, the number of RFs is incremented. This can be observed in Fig. 9(a), when encoding the frames 3 and 4 with two RFs and three RFs, respectively. Since the performance obtained for the frame 5 with four RFs was above the threshold limit, the *load balancing* was invoked to determine the distributions for encoding the frame 6 with the same number of RFs [see the arrow in Fig. 9(a)]. Since the achieved performance was below the threshold limit, these distributions were reused when encoding the frame 7 with five RFs. As before, obtaining the performance above the threshold, resulted in invoking the *load balancing* for encoding the frame 8 with the same number of RFs (5). Since the obtained performance for subsequent interframes was within the threshold region, the determined distributions were reused without altering the number of RFs. The only exception occurred for the frame 23, where a sudden change in the platform resulted in obtaining the encoding performance above the threshold. Hence, the *load balancing* was invoked for the frame 24, which allowed the real-time encoding by adapting to the current state of the platform.

Fig. 9(b) presents the achieved performance with the proposed adaptive algorithm in the SysDD, when automatically tuning both the number of RFs and the SA size. During the encoding of seven initial interframes with the 32×32 SA, the previously described approach was applied. In brief, after the initial equidistant partitioning for the first interframe, the *load balancing* was invoked for the second interframe. These distributions were reused for encoding the frames 3 and 4 with two RFs and three RFs, respectively. Afterward, the distributions were recalculated for the frame 5 and reused for the frame 6 with the increased number of RFs (4). In the proposed adaptive algorithm, whenever the real-time encoding is achieved with four RFs, the SA size is increased and the number of RFs is set to one. This methodology is adopted due to the similar ME computational load among these two encoding parameters. As presented in Fig. 9(b), after the real-time encoding was achieved for the frame 7 with four RFs, the SA size was increased to 64×64 and the determined distributions were reused for frame 8 with one RF. Since the obtained performance was above the upper threshold limit, the *load balancing* was invoked for frame 9. As the performance obtained for all subsequent interframes was within the threshold region, the distributions were reused without changing the number of RFs.

VII. Conclusion

In this paper, a collaborative interloop video encoding framework for CPU + GPU platforms is proposed. This framework relies on novel adaptive scheduling and load balancing methods to challenge a real-time H.264/AVC interloop encoding. These methods consider the overall interloop complexity, while exploiting several levels of parallelism and computation/communication concurrency. The library with highly optimized algorithms for all interloop modules was developed to support a wide range of CPU and GPU architectures. The proposed framework includes simultaneous execution control for different devices and automatic data access management to minimize the communication volume. This framework relies on the dynamic performance characterization of devices and communication links to iteratively improve the load balancing decisions. The proposed adaptive methods allow adjusting the performance and encoding parameters according to the current state of the platform and the real-time limit.

The framework efficiency was experimentally evaluated in several CPU + GPU platforms with different capabilities. The real-time encoding of 1080p sequences was obtained for challenging encoding parameters, i.e., 64×64 SA and/or multiple RFs. The proposed framework outperforms single-device executions for several times, while delivering the performance improvements of up to 61.2% over the state-of-the-art solution. The main directions for future work include support for other video coding standards (e.g., HEVC/H.265), a wider set of devices and systems, and the automatic tuning of encoding parameters to achieve the maximum coding efficiency.

Appendix
Determination of Additional/Remaining Transfers

The procedure presented herein determines the number and the amount of additional and remaining transfers when the load distributions for different divisible applications (modules) refer to the data located in the same buffer.

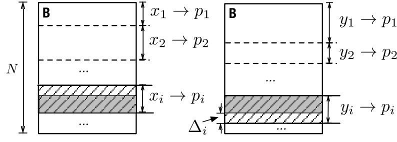

Fig. 10. Example of two application sharing the access to a buffer B.

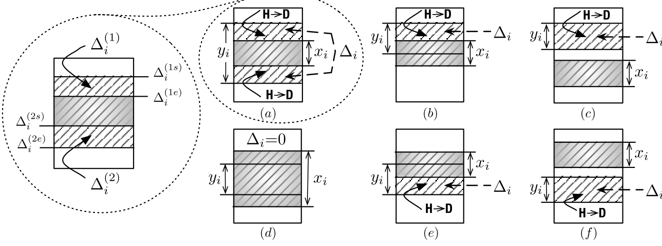

Fig. 11. (a)–(f) All possible additional transfers in the shared buffer.

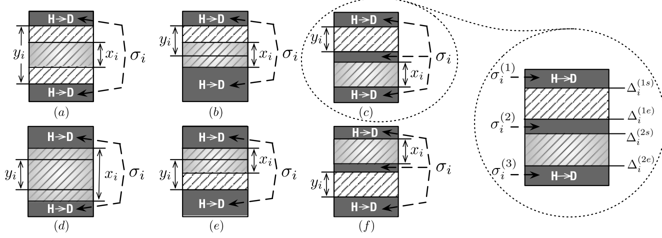

Fig. 12. (a)–(f) All possible remaining transfers in the shared buffer.

Let us consider two divisible applications $\mathbb{X}$ and $\mathbb{Y}$ with the total amount of load N to be distributed among the p_i accelerators in a heterogenous system, where $i \in \{1, \ldots, n\}$. Both applications share the access to a single data buffer B and their execution order is strictly defined, i.e., $\mathbb{Y}$ is executed after $\mathbb{X}$. Let $x = \{x_i\}$ and $y = \{y_i\}$ be the distribution vectors with the amount of loads from B to be processed on each p_i device for $\mathbb{X}$ and $\mathbb{Y}$, respectively (where $\sum x_i = N$ and $\sum y_i = N$).

To process the assigned x_i and y_i distributions, the corresponding data from the B buffer must be explicitly transferred to the p_i accelerator (see the areas with stripes in Fig. 10). Since these distributions address the same buffer, there is an opportunity to minimize the communication volume in cases when x_i and y_i distributions refer to the same locations. In detail, the y_i distribution may refer to the data portion that is already transferred for the x_i (see the dark areas in Fig. 10). Hence, due to the strict order of applications, only the portion of y_i needs to be transferred (see the Δ_i part in Fig. 10). This portion of data is referred herein as the *additional transfer*.

The amount and the number of additional transfers for p_i accelerator depend on the x_i and y_i distributions and on their stating and ending positions in the B buffer. The starting positions are calculated as $\sum_{j=1}^{i-1} x_j$ and $\sum_{j=1}^{i-1} y_j$, where x_j and y_j represent the distributions assigned to all previously enumerated devices ($j < i$). Correspondingly, the ending positions are calculated as $\sum_{j=1}^{i} x_j$ and $\sum_{j=1}^{i} y_j$.

Fig. 11 presents all possible *additional transfers* (Δ_i) within the B buffer according to the relative disposition of

Algorithm 3 LP_MIN

Input: χ_1, χ_2
Output: μ

$$\mu \le \chi_1, \mu \le \chi_2 \tag{28}$$
$$\mu \ge \chi_1 - N(1 - \hat{\chi}_1) \tag{29}$$
$$\mu \ge \chi_2 - N(1 - \hat{\chi}_2) \tag{30}$$
$$\hat{x}_1 + \hat{\chi}_2 = 1 \tag{31}$$
$$\hat{\chi}_1 \in \{0, 1\}, \hat{\chi}_2 \in \{0, 1\}$$

the x_i and y_i distributions and their starting/ending positions. As it can be observed in Fig. 11(a), at most two *additional transfers* can be performed, i.e., $\Delta_i^{(1)}$ and $\Delta_i^{(2)}$. The amount of these transfers is determined in respect of their starting and ending positions, i.e., $\Delta_i^{(1)} = \Delta_i^{(1e)} - \Delta_i^{(1s)}$ and $\Delta_i^{(2)} = \Delta_i^{(2e)} - \Delta_i^{(2s)}$. The starting and ending positions are calculated as

$$\Delta_i^{(1s)} = \min\left\{\sum_{j=1}^{i-1} x_j, \sum_{j=1}^{i-1} y_j\right\};\ \Delta_i^{(1e)} = \min\left\{\sum_{j=1}^{i-1} x_j, \sum_{j=1}^{i} y_j\right\} \tag{26}$$

$$\Delta_i^{(2s)} = \max\left\{\sum_{j=1}^{i} x_j, \sum_{j=1}^{i-1} y_j\right\};\ \Delta_i^{(2e)} = \max\left\{\sum_{j=1}^{i} x_j, \sum_{j=1}^{i} y_j\right\}. \tag{27}$$

Finally, the total amount of the *additional transfers* is computed as $\Delta_i = \Delta_i^{(1)} + \Delta_i^{(2)}$.

To retrieve the full content of the B buffer at the p_i accelerator, i.e., to complete the buffer, several remaining transfers need to be performed. As before, in certain cases, it is possible to minimize the overall communication volume for this procedure. The amount and the number of *remaining transfers* depend on the relative disposition of the x_i and y_i distributions, their starting and ending positions, as well as on the performed *additional transfers*.

Fig. 12 shows all possible *remaining transfers* (σ_i) within the shared B buffer. As it can be observed in Fig. 12(c), at most three *remaining transfers* can be performed, i.e., $\sigma_i^{(1)}$, $\sigma_i^{(2)}$, and $\sigma_i^{(3)}$. The amount of these transfers is determined in respect of the assigned distributions and the starting/ending positions of the performed additional transfers, i.e., $\sigma_i^{(1)} = \Delta_i^{(1s)}$, $\sigma_i^{(2)} = \Delta_i^{(2s)} - \Delta_i^{(1e)} - x_i$, and $\sigma_i^{(3)} = N - \Delta_i^{(2e)}$. As a result, the total amount of the *remaining transfers* is calculated as $\sigma_i = \sigma_i^{(1)} + \sigma_i^{(2)} + \sigma_i^{(3)}$. The complete analysis regarding the determination of the additional and remaining transfers can be found in [5].

The crucial point in determining the total amount of additional and remaining transfers lies in finding the minimum and the maximum between two values in linear programming. These procedures are presented as LP_MIN and LP_MAX in Algorithms 3 and 4, respectively. The minimum/maximum μ value between χ_1 and χ_2 is found by relying on auxiliary binary variables $\hat{\chi}_1$ and $\hat{\chi}_2$ and the total number of loads N as a big-M coefficient [18]. For example, if χ_i is the

Algorithm 4 LP_MAX

Input: χ_1, χ_2
Output: μ

$$\mu \geq \chi_1, \mu \geq \chi_2 \quad (32)$$
$$\mu \leq \chi_1 + N(1 - \hat{\chi}_1) \quad (33)$$
$$\mu \leq \chi_2 + N(1 - \hat{\chi}_2) \quad (34)$$
$$\hat{\chi}_1 + \hat{\chi}_2 = 1 \quad (35)$$
$$\hat{\chi}_1 \in \{0, 1\}, \hat{\chi}_2 \in \{0, 1\}$$

Algorithm 5 LP_BOUNDS Procedure

Input: $x = \{x_i\}, y = \{y_i\}$
Output: $\{\Delta_i, (\Delta_i^{(1)}, \Delta_i^{(2)}), (\Delta_i^{(1s)}, \Delta_i^{(1e)}, \Delta_i^{(2s)}, \Delta_i^{(2e)}) \mid i = \{1, \ldots, n\}\}$

$$\Delta_1^{(1s)} = \Delta_1^{(1e)} = 0, \Delta_1^{(2s)} = x_1, \Delta_1^{(2e)} = \text{LP_MAX}(x_1, y_1) \quad (36)$$
$\forall i \in \{2, .., n_w\}$:
$$\Delta_i^{(1s)} = \text{LP_MIN}(\textstyle\sum_{j=1}^{i-1} x_j, \sum_{j=1}^{i-1} y_j) \quad (37)$$
$$\Delta_i^{(1e)} = \text{LP_MIN}(\textstyle\sum_{j=1}^{i-1} x_j, \sum_{j=1}^{i} y_j) \quad (38)$$
$$\Delta_i^{(2s)} = \text{LP_MAX}(\textstyle\sum_{j=1}^{i} x_j, \sum_{j=1}^{i-1} y_j) \quad (39)$$
$$\Delta_i^{(2e)} = \text{LP_MAX}(\textstyle\sum_{j=1}^{i} x_j, \sum_{j=1}^{i} y_j) \quad (40)$$
$$\Delta_i^{(1)} = \Delta_i^{(1e)} - \Delta_i^{(1s)}, \Delta_i^{(2)} = \Delta_i^{(2e)} - \Delta_i^{(2s)}, \Delta_i = \Delta_i^{(1)} + \Delta_i^{(2)} \quad (41)$$

minimum value, the $\hat{\chi}_i$ binary variable is set to 1, while it is set to 0 otherwise. The binary variables and big-M coefficients are introduced to cover a full set of objective functions, especially when performing minimization over the minimum function (Algorithm 3) or maximization over the maximum function (Algorithm 4). In brief, when maximizing μ in LP_MIN, condition (28) is sufficient to guarantee that μ will take the value of the minimum between χ_1 and χ_2, since μ will be maximized until reaching the desired minimum. However, when minimizing μ in LP_MIN, depending on the other conditions in the linear program, μ can take an arbitrary value between 0 and the minimum between χ_1 and χ_2. Since the goal is to determine the exact minimum between two values, it is further needed to reinforce this condition by introducing $\hat{\chi}_1$ and $\hat{\chi}_2$ binary variables and N as the big-M coefficient in (29) and (30). As a result, only one binary variable will be set to 1 to designate the selected minimum (31). Similar rationale is applied for LP_MAX, where (32) is sufficient when minimizing μ, while (33)–(35) must be considered when maximizing μ.

By relying on the derived LP_MIN (Algorithm 3) and LP_MAX (Algorithm 4) procedures, the starting and ending positions of the additional transfers can be computed according to (26) and (27). To that respect, Algorithm 5 presents the LP_BOUNDS procedure, where these starting and ending positions are determined with conditions (36)–(40). In (41), the total amount (Δ_i) and the amount of each additional transfer ($\Delta_i^{(1)}$ and $\Delta_i^{(2)}$) are also computed. The LP_BOUNDS procedure is used in the proposed *Load Balancing* routine via MS_BOUNDS and LS_BOUNDS procedures (Algorithm 2). In detail, MS_BOUNDS procedure determines the additional transfers when ME and SME modules access CF or MV_ME buffers, thus $m = \{m_i\}$ and $s = \{s_i\}$ distribution vectors are used as input parameters. The LS_BOUNDS procedure determines the additional transfers when INT and SME modules share the access to SF, thus $l = \{l_i\}$ and $s = \{s_i\}$ distribution vectors are used as the inputs.

References

[1] Cisco, "Cisco visual networking index: Forecast and methodology, 2012–2017," Cisco, San Jose, CA, USA, Tech. Rep., May 2013.

[2] 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, Apr. 2004.

[3] G. J. Sullivan, J.-R. 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, Dec. 2012.

[4] 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, Jun. 2013, DOI: 10.1007/s11554-013-0357-y.

[5] A. Ilic, S. Momcilovic, and L. Sousa, "Scheduling and load balancing for multi-module applications on heterogeneous systems," INESC-ID, Lisbon, Portugal, Tech. Rep. 3/2015, Feb. 2014.

[6] 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.*, vol. 9, no. 1, pp. 5–18, Jan. 2013.

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

[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*, vol. 7131. Berlin, Germany: Springer-Verlag, 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 *Proc. IEEE 9th Int. Conf. High Perform. Comput. Commun., IEEE 14th Int. Conf. Embedded Softw. Syst. (HPCC-ICESS)*, Jun. 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," *Proc. SPIE*, vol. 7443, p. 74430X, Sep. 2009.

[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. Multimedia*, vol. 16, no. 1, pp. 108–121, Jan. 2014.

[12] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa, "Collaborative inter-prediction on CPU+GPU systems," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2014, pp. 1228–1232.

[13] A. Ilic, S. Momcilovic, N. Roma, and L. Sousa, "FEVES: Framework for efficient parallel video encoding on heterogeneous systems," in *Proc. 43rd Int. Conf. Parallel Process. (ICPP)*, Sep. 2014, pp. 20–29.

[14] 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. Appl.*, vol. 20, no. 3, pp. 365–381, 2006.

[15] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Appl. (ISPA)*, Jul. 2012, pp. 683–690.

[16] S. Momcilovic, A. Ilic, N. Roma, and L. Sousa, "Advanced video coding on CPUs and GPUs: Parallelization and RD analysis," INESC-ID, Lisbon, Portugal, Tech. Rep. 5/2013, Feb. 2013.

[17] 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, Mar. 2010.

[18] G. Sierksma, *Linear and Integer Programming: Theory and Practice*. Boca Raton, FL, USA: CRC Press, 2001.

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

[20] T. K. Tan, G. Sullivan, and T. Wedi, *Recommended Simulation Common Conditions for Coding Efficiency Experiments—Revision 3*, ITU, VCEG, document VCEG-AI10, Jul. 2008.

Aleksandar Ilic (M'14) received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon, Portugal, in 2014.

He is an Assistant Professor with the Department of Electrical and Computer Engineering, IST, and a Senior Researcher with the Signal Processing Systems Group, Instituto de Engenharia de Sistemas e Computadores—Investigação e Desenvolvimento, Lisbon. He has contributed over 20 papers in journals and international conferences. His research interests include high-performance and energy-efficient computing, and modeling on parallel heterogenous systems.

Svetislav Momcilovic (M'11) received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, in 2011.

He is a Senior Researcher with the Signal Processing Systems Group, Instituto de Engenharia de Sistemas e Computadores—Investigação e Desenvolvimento, Instituto Superior Técnico. He has contributed over 20 papers in journals and international conferences. His research interests include parallel computing, multicore architectures, heterogeneous parallel systems, and video coding.

Nuno Roma (SM'13) received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade de Lisboa, Lisbon, Portugal, in 2008.

He is an Assistant Professor with the Department of Electrical and Computer Engineering, IST, and a Senior Researcher with the Signal Processing Systems Group, Instituto de Engenharia de Sistemas e Computadores—Investigação e Desenvolvimento, Lisbon. He has contributed over 70 papers in journals and international conferences. His research interests include specialized computer architectures for digital signal processing, parallel processing, and high-performance computing.

Dr. Roma is also a Senior Member of the IEEE Circuits and Systems Society and a member of the Association for Computing Machinery.

Leonel Sousa (SM'03) received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, in 1996.

He is a Full Professor with Universidade de Lisboa and a Senior Researcher with the Instituto de Engenharia de Sistemas e Computadores—Investigação e Desenvolvimento, Lisbon. He has contributed over 200 papers in journals and international conferences. He has also contributed to the organization of several international conferences, as the Program Chair and the General and Topic Chair, and has given keynotes in some of them. His research interests include very-large-scale integration architectures, computer architectures, parallel computing, computer arithmetic, and signal processing systems.

Dr. Sousa is a fellow of the Institution of Engineering and Technology, a Senior Member of the Association for Computing Machinery, and a member of International Federation for Information Processing WG10.3 on concurrent systems. He received several awards for his contributions to more than 200 papers in journals and international conferences. He has edited two special issues of international journals. He is currently an Associate Editor of IEEE TRANSACTIONS ON MULTIMEDIA, IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, and *Journal of Real-Time Image Processing* (Springer), and the Editor-in-Chief of *EURASIP Journal on Embedded Systems*.