J Real-Time Image Proc (2016) 11:571–587
DOI 10.1007/s11554-013-0357-y

ORIGINAL RESEARCH PAPER

Exploiting task and data parallelism for advanced video coding on hybrid CPU + GPU platforms

Svetislav Momcilovic · Nuno Roma · Leonel Sousa

Received: 9 July 2012 / Accepted: 14 May 2013 / Published online: 2 June 2013

Abstract Considering the prevalent usage of multimedia applications on commodity computers equipped with both CPU and GPU devices, the possibility of simultaneously exploiting all parallelization capabilities of such hybrid platforms for high performance video encoding has been highly quested for. Accordingly, a method to concurrently implement the H.264/ advanced video coding (AVC) inter-loop on hybrid GPU + CPU platforms is proposed in this manuscript. This method comprises dynamic dependency aware task distribution methods and real-time computational load balancing over both the CPU and the GPU, according to an efficient dynamic performance modeling. With such optimal balance, the set of rather optimized parallel algorithms that were conceived for video coding on both the CPU and the GPU are dynamically instantiated in any of the existing processing devices, to minimize the overall encoding time. The proposed model does not only provide an efficient task scheduling and load balancing for H.264/AVC inter-loop, but it also does not introduce any significant computational burden to the time-limited video coding application. Furthermore, according to the presented set of experimental results, the proposed scheme has proved to provide speedup values as high as 2.5 when compared with highly optimized GPU-only encoding solutions or even other state of the art algorithm. Moreover, by simply using the existing computational resources that usually equip most commodity computers the proposed scheme is able to achieve inter-loop encoding rates as high as 40 fps at a HD 1920 $\times$ 1080 resolution.

Keywords Video coding · GPU · Hybrid CPU + GPU Platforms · Load balancing · CUDA

S. Momcilovic (✉) · N. Roma · L. Sousa
INESC-ID IST-TU Lisbon, Rua Alves Redol, 9, 1000-029 Lisbon, Portugal
e-mail: Svetislav.Momcilovic@inesc-id.pt

N. Roma
e-mail: Nuno.Roma@inesc-id.pt

L. Sousa
e-mail: Leonel.Sousa@inesc-id.pt

1 Introduction

The increasing demand for high-quality video communication and the tremendous growth of video contents on Internet and local storages stimulated the development of highly efficient compression methods over the past decades. When compared with previous standards, the H.264/ MPEG-4 advanced video coding (AVC) [1] achieves compression gains of about 50 %, keeping the same quality of the reconstructed video [2]. However, such compression efficiency comes at the cost of a dramatic increase of the involved computational requisites, making real-time video coding hard to be achieved even on the most recent single-core central processing units (CPUs).

The latest generations of commodity computers, equipped with both multi-core CPUs and many-core graphics processing units (GPUs), already offer high computing performances to execute a broad set of signal processing algorithms. In particular, the GPU architectures consist of hundreds of cores especially adapted to exploit fine-grained parallelism, and as such are frequently applied to implement complex signal processing applications. On the other hand, on the multi-core CPUs the data-parallelism can be exploited either at a coarse grained level, by concurrently running multiple threads on different cores, or at a fine-grained level, by using vector instructions. Hence, the simultaneous exploitation of all these different parallelization models, involving both the CPUs and the GPUs, leads to complex, but rather promising, challenges that are widely attractive and worth to be exploited by the most computationally demanding applications. However, even though these devices are able to run asynchronously, new and efficient parallelization models are needed to maximally exploit the computational power offered by these concurrently running devices. Nevertheless, such models must assure the inherent data dependencies in the parallelized algorithm, as well as a load-balanced execution on the processing devices.

The H.264/AVC [1] standard represents one of the most efficient video coding paradigms based on motion-compensated prediction. In the H.264/AVC, each frame is divided in multiple Macroblocks (MBs), which are subsequently encoded using either *inter* or *intra* prediction modes (see Fig. 1). In the most computationally demanding and most frequently applied *inter* mode, the best-matching predictor of each MB is searched within a restricted set of already encoded reference frames (RFs). This process, denoted by motion estimation (ME), is based on a further division of each 16×16 pixels MB into 7 different partitioning modes, namely, 16×8, 8×16, 8×8, 8×4, 4×8 and 4×4 pixels. The search procedure is then further refined by using the interpolated RFs with half-pixel and quarter-pixel precision. Then, an integer transform is applied to the residual signal, which is subsequently quantized and entropy coded, before it is sent to the decoder, alongside with the corresponding motion-vector information. The decoding process, composed of the dequantization, the inverse integer transform and deblocking filtering, is also implemented in the feedback loop of the encoder, to reconstruct the encoded RFs. This reconstruction is finalized by the application of a deblocking filter, to mitigate blocking artifacts that were introduced by the encoding process.

Recently, several proposals have been presented to implement parallel video encoding algorithms by using GPU devices [3–9]. A promising set of adaptive ME algorithms for NVIDIA GPUs was presented by Lu and Hang [3] and Schwalb et al. [4]. These algorithms mainly feature a reduction of the algorithm complexity, but still keep a high parallelization potential. However, the provided evaluation results only consider very low resolutions, which does not guarantee their efficiency for high-definition (HD) video sequences in what concerns both the performance and the encoding quality. Moreover, such results are not directly compared with other state of the art parallel GPU algorithms that have been proposed for the H.264/AVC. Furthermore, all these proposals mostly exploit data-level parallelism and most of them only focus on a single module of the prediction-loop of the H.264/AVC encoder.

Meanwhile, Cheung et al. [10] referred to the task partition between the CPU and the GPU as the main challenge for efficient video coding on commodity computers. At this respect, a computational model for scalable H.264 video coding for embedded many-core devices was proposed by Azevedo et al. [11]. These authors proposed the application of scheduling techniques at the MB level. For such purpose, a lightweight scheduling was achieved by using a specialized hardware for the optimized task submission/retrieval unit. However, the implementation of the scheduling control strategy that was considered for each MB makes this model rather expensive for CPU + GPU platforms.

A different approach for parallel implementations of the entire H.264/AVC encoder on CPU + GPU systems was proposed by Chen and Hang [12]. This approach completely offloads the ME task (including the interpolation and sub-pixel ME) to the GPU, keeping the rest of the modules to be executed on the CPU. However, this approach does not consider the possibility of concurrent and asynchronous implementation of the video coding algorithm on both the CPU and the GPU devices, which could potentially decrease the overall processing time. To

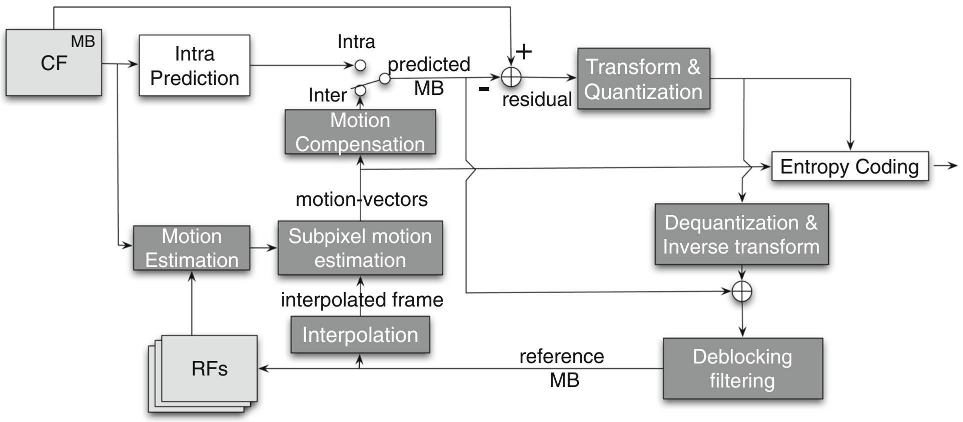


Fig. 1 Block diagram of the H.264/AVC encoder: *inter* encoding loop

the best of the authors' knowledge, there is still not any proposal that effectively considers a hybrid co-design parallelization approach, where the whole encoder structure is simultaneously scheduled on the CPU and on the GPU.

By taking this idea in mind, an entirely new parallelization approach is proposed herein. Such approach is based on a novel dynamic performance model for parallel implementation of the entire *inter* loop of the encoder that simultaneously and dynamically exploits the CPU and the GPU platforms. To optimize such a hybrid parallelization, a method for dynamic task and data distribution is proposed.

The proposed method relies on a realistic performance model that is built at run-time and improved in each iteration of the algorithm, to capture the real system behavior and automatically discover the performance of the devices. Even though non-dedicated or non-reliable systems are not the main target of this work, the proposed dynamic performance model also allows an automatic correction of perturbations or eventual performance variations due to external causes that may occur in real systems. Moreover, it exploits several parallelization levels currently available in these systems, ranging from the fine-grained thread-level parallelism on the GPU, to both thread and vector-level parallelization on the CPU side.

In contrast to Momcilovic et al. [13], this paper provides a complete parallelization strategy comprising all *inter*-frame processing modules of the H.264/AVC encoder, considering both CPU and GPU devices, as well as an improved scheduling algorithm for collaborative video encoding on hybrid GPU + CPU platforms. The proposed parallel algorithms have been developed both in the Compute Unified Device Architecture (CUDA) for GPU platforms, and by using the OpenMP + SSE for the multi-core CPU. The parallelized modules implement the full-pixel ME, interpolation, sub-pixel motion estimation (SME), quantization, integer transform, inverse integer transform, dequantization and the deblocking filtering. However, it should be noted that the achieved parallelizations of the H.264/AVC *inter*-loop modules on both CPU and GPU does not only require different strategies caused by the core granularity, but they also demand completely different programming and parallelization approaches, according to the existing differences in the architectures, programming environments and applied programming models. The scheduling scheme proposed herein additionally provides a more accurate evaluation of the achieved performance, including an initial set of test frames, and proposes an entirely new strategy of mode-based parallel SME for hybrid GPU + CPU platforms.

This manuscript is organized as follows: in Sect. 2, the dependency issue existing in the H.264/AVC encoder is analyzed, and the profiling of the encoder on both the CPU and the GPU is presented. The dynamic model for parallel video coding on CPU + GPU platforms is proposed in Sect. 3. Section 4 describes the conceived parallel implementation of the H.264/AVC encoding algorithm in this hybrid platform. The evaluation of the experimental results is presented in Sect. 5, while Sect. 6 summarizes the main contribution of the presented research.

2 H.264/AVC data dependencies and profiling analysis

Before parallel processing and balanced load distribution techniques can be considered to speedup the H.264/AVC video encoder, an extensive analysis of inherent data dependencies needs to be performed. In the H.264/AVC encoder, two classes of dependencies can be identified, namely, data dependencies and control dependencies. Furthermore, data dependencies can still be classified in three distinctive subclasses in this specific algorithm, namely, *inter*-frame dependencies, *intra*-frame dependencies and the inherent sequential dependencies between the H.264/AVC modules.

Inter-frame data dependencies exist between different frames in the video sequence. Due to these dependencies, the encoding of one frame cannot be performed before the encoding of some other frame has already finished. Such dependencies are mainly present in *inter*-frame prediction (P and B type frames), between the current and the considered RFs. In this prediction mode, motion is estimated by finding the best matching between the pixels in the current frame and the reconstructed pixels from already coded and transmitted RFs. As a consequence, all RFs must be decoded in the reconstruction loop of the encoder before the ME procedure can be initiated.

Intra-frame data dependencies are present in the processing of neighboring areas of the same frame. Although such dependencies predominantly exist between adjacent blocks in *intra* (I) type frames, they are also observed in *inter* (P and B) frames, namely, in the prediction of the motion vectors by considering the neighboring MBs (when computing the median vector to determine the starting search point), or in the deblocking filtering (when the MB edges are filtered).

Dependencies between the H.264/AVC modules indicate that the output data of one processing module corresponds to the input data of another subsequent module, as depicted in Fig. 1. For example, the motion vectors correspond to the output of the ME module and together with the sub-pixels values that are computed by the interpolation module define the initial search point and the search space for the subsequent SME module. Nevertheless, the interpolation and the ME modules do not need to wait for each other, since both of them use the current frame and/or the RFs.

Control dependencies are present whenever data-related conditions may imply different processing branches for the input data. For example, in the case of the deblocking filtering, different edges (and even different distance values from the edges) may cause different filtering procedures.

To assure the compliance of the whole set of strict dependencies imposed by the H.264/AVC encoding algorithm, it can be observed that an efficient parallel processing implementation can only be considered at a single frame level, since the *inter* prediction procedure cannot start before the list of all considered RFs is created/updated. Moreover, due to the set of *intra*-frame dependencies that has to be fulfilled in the deblocking filter, this module cannot be divided in independent and concurrent parts. Consequently, these two observations exclude the possibility of splitting each frame in several parts to be simultaneously processed in a pipeline and concurrent fashion. Hence, by considering the existing data dependencies between the several H.264/AVC processing modules, a dependency diagram of the encoder *inter*-loop can be defined, as it is presented in Fig. 2. As it can be seen, the interpolation and the full-pixel ME modules can be processed in parallel. However, the SME can only start when these two procedures are completed. Finally, since the rest of the modules are also dependent on each other, i.e., the output of each module is the input of the following one, they will have to be sequentially processed.

Due to its complexity dominance in the whole encoding procedure, the particular case of the ME module is worth noting. As it was previously referred, the *intra*-frame dependencies that are observed in this module seriously limit the possibility of data level parallelization. Namely, to find the best matching for the current MB in the considered RFs, the search is usually performed in an area around the position of the current MB displaced by an offset that is defined by the median vector. This vector is computed as a median value of the set of motion vectors computed for its left, up, and right-up neighbors. However, it was already shown in [5] that when exhaustive search is performed, the resulting penalty of using the zero motion vector as the central search position is not significant. Therefore, to allow a finer grained parallelization, the adoption of the exhaustive Full-Search Block-Matching (FSBM) algorithm is worth to be considered, by assuming a zero motion vector candidate as the search area center. It is also worth mentioning that the high computational load required by this exhaustive search algorithm can be significantly reduced by hierarchically computing the sum of absolute differences (SAD) distortion measure, i.e., by reusing the results obtained for smaller partitions of the MB under consideration [14].

Finally, it can be observed from Fig. 1 that the entropy coding module only depends on the quantization module, since it uses the quantized coefficients as its input. On the other hand, the output of this module represents the output video stream and does not impose any feedback to the remaining H.264/AVC encoding modules. Consequently, the processing of the entropy coding module can be completely overlapped with the *inter*-loop modules, and therefore it does not limit the overall performance as long as its execution time is shorter than the execution time of the entire *inter*-loop, which is generally verified.

Figure 3a presents the partitioning of the H.264/AVC processing time in a GPU device implementation, regarding to the various AVC modules. These results were measured by considering highly optimized GPU algorithms, which will be presented in more detail in Sect. 4. The considered resolution for the test video sequences was 1280 × 720 pixels and the used GPU device was an (MSI) NVIDIA GeForce GTX 580. Similar results were obtained for other GPUs and using several other resolutions for the test video sequences. As it can be seen, from the computational complexity point of view, the full-pixel ME is a dominant module, with more than 75 % of the total processing time.

Contrasting with the interpolation module, which participates similarly (about 4 %) to the total processing time at both devices, the deblocking filtering is relatively more demanding in the case of the GPU implementation. This is mainly due to the limited parallelization potential of this module, caused by strict *intra*-frame and control dependencies, as it was mentioned above. Finally, it is also observed that the remaining modules, namely, motion compensation, integer transform, quantization, dequantization and inverse

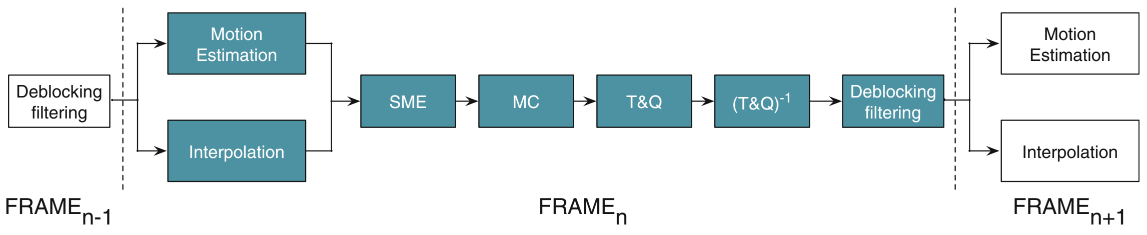
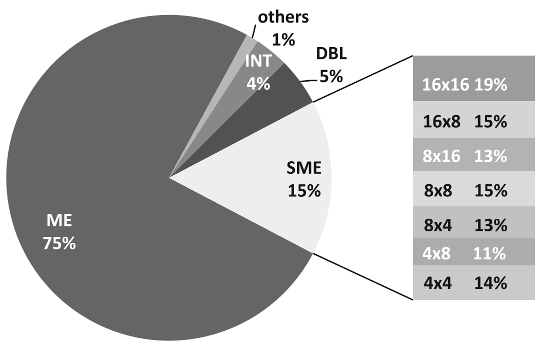
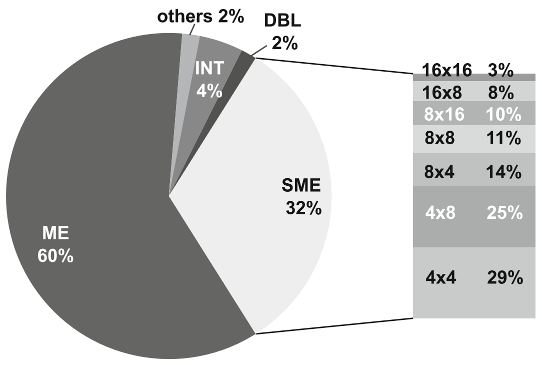


Fig. 2 Dependencies between the several processing modules in a H.264/AVC video encoder



(a) GPU implementation.



(b) CPU implementation.

Fig. 3 Partitioning of the H.264/AVC *inter* prediction loop processing time (*ME* full-pixel motion estimation, *SME* sub-pixel motion estimation, *INT* interpolation, *DBL* deblocking filter, *Others* direct integer transform, quantization, dequantization and inverse integer transform)

transform, only represents about 1–2 % of the total processing time.

The processing time of the SME module is presented with a greater detail on the right side of Fig. 3a, by considering seven different kernels corresponding to the different MB partitioning modes (see further details in Sect. 3.3). It can be seen that the most of the computation time devoted to this module is spent in the processing of the 16 × 16 mode, i.e. when the MB is not divided and the refinement is performed for only one single partition. This is mainly due to the fact that in this MB partitioning mode the number of concurrently processed partitions is smaller, leading to less efficient implementations in the GPUs, primarily designed for massive exploitation of fine-grain parallelism.

Figure 3b represents the partitioning of the processing time of the *inter* prediction loop in a general purpose CPU, regarding to the same video coding modules. These results were obtained for highly optimized code in a Intel Core i7 950 processor Int (2008). Both core level parallelization (using OpenMP API) and vectorization (using SSE ISA extensions) were extensively performed. By comparing with the GPU implementation, it is possible to observe that ME has a slightly lower weight in the processing time. On the other hand, the SME module takes a significantly larger part in the overall time. However, and contrasting to what happens in the GPU, it is the set of SME modes with finer MB partitioning that requires more computation time. Finally, it is also observed that the percentage of the processing time devoted to the deblocking filter is significantly smaller, when compared with the equivalent GPU implementation.

3 Performance model and task distribution for parallel coding on a hybrid CPU + GPU system

Considering the dependency analyzes conducted in Sect. 2, it is observed that the heterogeneous structure of the H.264/AVC encoder includes several modules without any data-dependency when processing neighboring pixels (e.g., quantization), but also highly data-dependent modules (e.g., deblocking filtering). The suitability of all these algorithms for fine-grained or coarse-grained parallelization is also highly variable and some of them are more efficiently implemented on one or another processing device. In this section, it is analyzed the possibility of minimizing the video encoding time by efficiently distributing the encoding tasks on the CPU and the GPU devices. For such purpose, it will be proposed a new dynamic performance model for parallel video encoding by considering a hybrid co-design parallelization approach, where the CPU and the GPU are simultaneously and asynchronously scheduled, to speedup the execution of the whole encoding structure.

3.1 Load distribution and scheduling strategy

From the profiling analysis presented in Sect. 2, it is clear the potential advantage of distributing the most computationally demanding modules, namely ME and SME, among the CPU and the GPU devices. In particular, the ME module can be simultaneously implemented by both devices on different parts of the frame, by considering a frame division at the level of rows/columns of MBs. This distribution should be performed in a rather dynamic way, by taking into account the performance level that is offered by each device, evaluated during the processing of the previous encoded frame. Since the interpolation can be simultaneously executed with the ME, this module is also considered when distributing the ME operation. Nevertheless, a different processing scheme is adopted for the SME module, due to the fact that each MB partitioning mode offers different paralellization potential and is rather implemented by a separate CUDA kernel. As a consequence, the load balance is performed by distributing the processing of the several prediction modes to the CPU and the GPU devices.

As soon as the most computationally intensive modules are assigned, the distribution of the remaining encoding modules is defined. For such purpose, a module-level scheduling is applied to the rest of the modules, in such a way that the overall processing time is minimized. Since all the other modules that can be concurrently processed (i.e., except the deblocking filter) only take about 1 % of the total time, their distribution among the CPU and GPU would not offer any significant advantage. Nevertheless, their execution is still evaluated in both processing devices (by using a predefined set of test frames), to predict further performance gains and to achieve an optimal module-level scheduling.

Algorithm 1 Scheduling/load balancing algorithm.

```
1: for frame_nr=0 to (nr_of_test_frames − 1) do
2:    load part of the frame on each device
3:    execute all the modules on the available devices
4:    update performance metrics
5: end for
6: for frame_nr= nr_of_test_frames to nr_of_frames − 1 do
7:    (re-)evaluate distribution for ME and interpolation
8:    (re-)evaluate distribution for SME
9:    (re-)evaluate distribution for the remaining modules
10:   in parallel on CPU and GPU do
11:      load part of the frame on each device
12:      execute part of ME on each device
13:      execute interpolation on the assigned device
14:      exchange results
15:      execute part of SME on each device
16:      exchange results
17:   end in parallel
18:   execute the remaining modules on the assigned device
19:   update performance metrics
20: end for
```

Algorithm 1 presents the implementation of the proposed method. A detailed formalization of the most complex steps will be presented in the following subsections. Along the encoding procedure, the video sequence under processing is divided in batches, comprising a pre-defined amount of frames. Within each batch, the proposed load balancing algorithm defines a restricted set of frames that will be used for profiling and dynamic adjustment of the load distribution and tasks assignment. Within this *profiling* stage, a restricted set of test frames (e.g., 3 frames) is encoded both on the CPU and on the GPU, by adopting an even distribution of the frame area among the several devices (lines 1–5). With the performance data that are collected at this stage, the load distribution algorithm will start assigning the several encoding modules and frame data in a balanced way, to minimize the resulting encoding time (lines 10–17).

Initially, the ME and the interpolation modules are jointly distributed, since these modules can be simultaneously processed (see Sect. 3.2). For such purpose, a given number of MB-columns is loaded to each of the devices. Such amount of MB-columns to be processed by each device will be updated over the time. As soon as both procedures are finished on both the CPU and the GPU, the resulting output values are exchanged, to allow the encoding algorithm to continue with the subsequent modules. Only then will the distribution of SME module be considered (see Sect. 3.3), to divide the several prediction modes among the available devices (line 15). At the end, the remaining modules are serially processed and assigned to a single device (line 18), as will be described in Sect. 3.4.

Throughout this dynamic load balancing method, the processing time of each of the encoding modules is constantly measured to evaluate their performance. The measured times are updated after processing each frame (line 19). This is particularly important for the most computationally intensive modules, due to the fact that any eventual change of the performance level on some of the devices (for example, the CPU, which will be also running the operating system) have a negative effect on the load balance, with a consequent increase of the processing time. For this distribution, both the processing time and any eventual data transfer time are considered (see Sect. 3.4). Therefore, this distribution is performed dynamically and permanently using these updated times.

In Fig. 4, it is represented as an example of a complete distribution of the H.264/AVC modules in a hybrid CPU + GPU system, by applying the proposed model. In this case, the ME and the SME modules are distributed to both processing devices, while the interpolation is assigned to the GPU. The rest of the modules are serially processed in these devices, to minimize the overall processing time. A typical situation is the one where the deblocking filtering is performed in the CPU side, due to the presence of numerous *intra*-frame and control dependencies, which eventually limit the efficiency of the GPU algorithm.

The computationally lightweight and not time consuming scheduling that is now proposed, as well as its corresponding load balancing strategy, are performed on a single CPU core. The frame data is distributed in the following way. The current frame is initially divided according to the predicted performance of the ME module on both the CPU

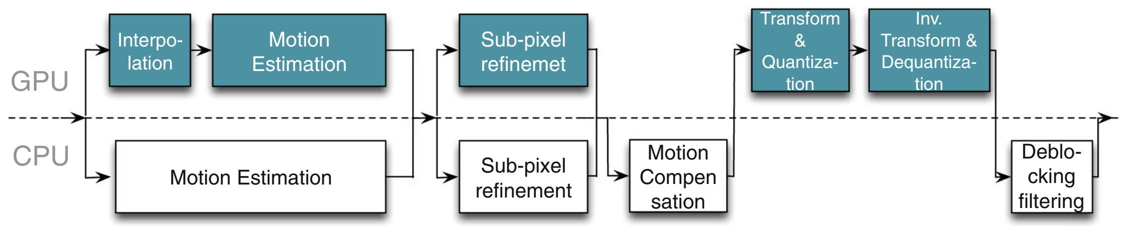


Fig. 4 Application of the proposed dynamic load balancing scheme to the whole H.264/AVC encoding loop, by using a single GPU and a single CPU

and the GPU (see Sect. 3.2), and the fraction of the frame that is assigned for execution on the GPU is immediately transferred. On the other hand, the data transfers corresponding to the remaining parts of the current frame, as well as of the interpolated frame are overlapped with the processing of the ME module, which is allowed by the ability of the GPU to overlap the computation with the data transfers. Finally, at the end of the inter-loop, the newest RF is transferred to the device that did not perform the deblocking filtering, which allows the model to keep the lists of the RFs updated on both devices.

Finally, although this model was specifically designed for video encoding, it is worth noting that it can equally be applied to any other data processing algorithm composed of several distinct tasks. Instead of frames to be encoded, any other data units can be considered and entirely similar optimizations steps should be applied to find the best distribution. Naturally, it is only worth to apply this distribution model if the CPU and the GPU implementations of the algorithm are competitive in the implementation of some of the tasks or sub-tasks that integrate the considered processing system, to efficiently exploit the computational and parallelization capabilities that are offered by these platforms.

3.2 Dynamic load balancing of the motion estimation and interpolation modules

Considering the availability of several processing devices for concurrent computing, the large computational load of the Full-Search Block-Matching (FSBM) motion estimation module is particularly fitted to be divided and distributed over the existing processing devices. The distribution is performed at the level of MB-columns, by sending n_{gpu} MB-columns to the GPU, and $(n - n_{\text{gpu}})$ MB-columns to the CPU, where n is a total number of MBs columns within a single frame (or slice). It is also assumed that the attained performance (s), expressed as the number of processed MB-columns per second, does not depend on the considered distribution, i.e. $s_{\text{gpu}} = c^{te}$ and $s_{\text{cpu}} = c^{te}$ within a single frame. Hence, the ME processing time in the two platforms can be expressed as:

$$t_{\text{gpu\_me}} = \frac{n_{\text{gpu}}}{s_{\text{gpu}}}, \quad t_{\text{cpu\_me}} = \frac{n - n_{\text{gpu}}}{s_{\text{cpu}}}. \tag{1}$$

The constant performance assumption that is considered here is mainly supported by the following two reasons. First, the relation between the required computation and the inherent data-transfer times is very high (up to two orders of magnitude), as it will be confirmed in the experimental results section (see Table 4). The second reason is allied to the fact that this model allows very simple and efficient distribution mechanisms that do not involve a significant computational burden.

Hence, the main aim of the proposed dynamic load balancing scheme is to find the optimal distribution of MB-columns that will provide the most balanced execution time in the two processing devices. Equation 2 defines such balanced distribution, where $t_{\text{gpu\_me}}$ and $t_{\text{cpu\_me}}$ represent the processing time of the ME module on the CPU and on the GPU, respectively. Conversely, t_{gpu0} and t_{cpu0} represent the processing time of the modules that are supposed to be executed in the CPU and in the GPU, together with the ME. As an example, such set of modules may include the interpolation operation, which does not have any data dependency with ME, and is required to be executed on one of these devices.

$$t_{\text{cpu\_me}} + t_{\text{cpu0}} \approx t_{\text{gpu\_me}} + t_{\text{gpu0}}, \tag{2}$$

By combining Eqs. 1 and 2 it is obtained:

$$\frac{n - n_{\text{gpu}}}{s_{\text{cpu}}} + t_{\text{cpu0}} \approx \frac{n_{\text{gpu}}}{s_{\text{gpu}}} + t_{\text{gpu0}}, \tag{3}$$

The processing workload on the GPU device will be given as:

$$n_{\text{gpu}} \approx \frac{n + s_{\text{cpu}}(t_{\text{cpu0}} - t_{\text{gpu0}})}{1 + \frac{s_{\text{cpu}}}{s_{\text{gpu}}}}. \tag{4}$$

The assignment of the interpolation module is done according to the ratio of the performances on the two devices (i.e. speedup). In particular, if the interpolation is predicted to have a larger speedup than the ME module when sending it from the CPU to the GPU, it is assigned a greater offloading priority. Otherwise, it remains in the CPU, while the ME is simultaneously performed on the

GPU. As soon as the interpolation is finished, the ME starts to be simultaneously executed on both devices.

However, the measured performance in any real system varies along the time, mainly due to inherent changes of the conditions it operates on (e.g., operation system). Moreover, in real situations it is often needed to re-evaluate performance to have more accurate distribution of the computational load among the devices. Therefore, if the number of MB-columns that is assigned to each device is determined by only considering the encoding time of a single frame, the obtained distribution will hardly be accurate along the time. As a consequence, the number of MB-columns that is submitted to the GPU should be updated in every iteration:

$$n^i_{\text{gpu}} \approx \frac{n - s^{i-1}_{\text{cpu}} \Delta t_0}{1 + \frac{s^{i-1}_{\text{cpu}}}{s^{i-1}_{\text{gpu}}}}, \quad n^i_{\text{cpu}} = n - n^i_{\text{gpu}} \tag{5}$$

where $\Delta t_0 = t_{\text{cpu0}} - t_{\text{gpu0}}$ is the signed sum of distribution-independent task portions on the CPU (positive sign) and on the GPU (negative sign). The measured performance values ($s^{i\text{-}1}_{\text{cpu}}$ and $s^{i\text{-}1}_{\text{gpu}}$) are updated upon the encoding of each frame, according to the measured ME processing time on both devices. Hence, this iterative procedure starts with a predicted value (e.g., $n^0_{\text{gpu}} = \#\text{MBcolumns}/2$) and is updated until it converges to the ideal distribution, based on the measured performance on both processing devices.

3.3 Dynamic load balancing of the sub-pixel refinement module

In contrast with the ME module, where the considered search strategy adopts the same initial search point for all the MB partitions [5], in the SME module the initial search point is defined by the best matching predictors that were found during the ME process for each partition. Due to the fact that all these partitions consider different search areas, a hierarchical distortion computation (by using re-usage schemes) is not viable. Therefore, there is no any significant advantage of using a single kernel on the GPU, and it is much more efficient to implement the SME module in seven different kernels, corresponding to the different MB partition modes [5]. The individual parallelization of each of these modes should be done according to their distinct parallelization potential in what concerns to the adopted processing granularity. In particular, finer granularity allows multiple partitions to be simultaneously processed, while less work will be performed within a single processing partitioning.

Hence, the existence of several independent functions that can be simultaneously processed opens the possibility to exploit function-level parallelism. As in the case of the interpolation and ME modules, the predicted speedup (see Sect. 3.2) that is evaluated for the SME module during the initial test frames is used as a preliminary parameter.

Algorithm 2 GPU sub-pixel motion estimation iteration on a single thread level.

```
8.0: if frame_nr==nr_of_test_frames
8.1:   $G_{sme} = \{sme_{m\times n}, m\times n = 16\times16 \,..\, 4\times4\}$; $C_{sme} = \{\emptyset\}$
8.2: else
8.3:   find $sme_{p\times q}$, such $\frac{s_c(p\times q)}{s_g(p\times q)} = max\{\frac{s_c(m\times n)}{s_g(m\times n)}\}$
8.4:   if $t_{Gsme} < t_{Csme}$
8.5:       $G_{sme} = G_{sme} \setminus sme_{p\times q}$; $C_{sme} = C_{sme} \cup sme_{p\times q}$
8.6:   else
8.7:       (re-)evaluate $sme_{p\times q}$ distribution among devices
8.9:   endif
8.9 endif
```

Algorithm 2 presents the sub-pixel ME load balancing procedure, which contains the sub-steps corresponding to line 8 of Algorithm 1. C_{sme} and G_{sme} represent the set of SME modes running on the CPU and on the GPU, respectively, while $sme_{m\,\times\,n}$ represents the corresponding mode of the $m \times n$ partition size. $s_c(p \times q)$ and $s_g(p \times q)$ refer to the performance values (in MB-columns per second) of the $p \times q$ mode, while t_{Csme} and t_{Gsme} correspond to the processing times needed to perform the partitioning modes of C_{sme} and G_{sme} sets, respectively, on the related devices. The $\cup$ and the $\setminus$ are the set union and the set difference operations, respectively.

Initially, all the kernels/modes are assigned to one of the devices (line 8.1). After that, the functions related to the different MB partitioning modes are offloaded one by one, giving priority to those that achieve the highest speedup on the related device (lines 8.3 and 8.5). As soon as the encoding time on the device that accepts the offloaded function is larger than the time on the originating device (line 8.4), the offloading process will stop and the final load balancing is achieved upon the distribution of this last offloaded function (line 8.7), as it was explained in Sect. 3.2. Just as before, Eq. 3 is applied along this optimization process, where t_{cpu0} represents the processing time of the remaining modules that are implemented in the CPU, while t_{gpu0} represents the processing time of all other modules that are processed in the GPU. At this respect, an additional and rather useful feature that is offered by the most recent NVIDIA GPU devices is the ability to adopt streaming processing schemes [16, 17]. In particular, whenever data independent functions of a given module are implemented in different CUDA kernels, this streaming processing model can be efficiently exploited by putting them in different CUDA streams. In such a way, the several modules under processing can pipeline the data transfers

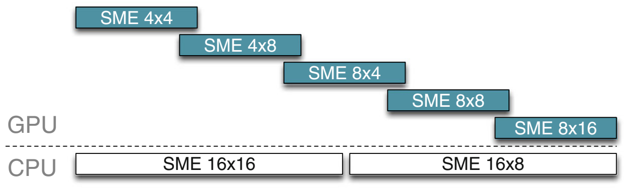


Fig. 5 Application of the proposed dynamic load balancing scheme to the sub-pixel motion estimation (SME) module

and the execution stages with each other. Moreover, whenever a given kernel does not occupy all the available streaming multiprocessors, two kernels can compete for the existing resources, which can further decrease the overall encoding time. In Fig. 5, it is represented a possible distribution of the SME modes on the CPU + GPU platform. The overlapping of the modules in the GPU side (in the horizontal direction) symbolically represents the overlapping of the computation and of the data-transfers in different GPU kernels. As in the case of ME, the performance of the data-transfers are also considered when computing the performance of the kernels/modes.

3.4 Scheduling of the remaining modules

As it was described in Sect. 3.1, the least computationally intensive modules of the encoder are scheduled to be completely executed on one of the processing devices. The same happens with the deblocking filter, whose implementation cannot be efficiently split by multiple devices. The proposed scheduling scheme is then applied, to minimize the overall processing time. For such purpose, all these modules are implemented and evaluated on both the CPU and GPU, and the measured processing times are then used as input parameters for the distribution algorithm, altogether with the data transfer times, required for any module transition between the devices.

The proposed distribution procedure is illustrated in Fig. 6. A data-flow diagram for all the encoding modules, considering both the CPU and GPU, is initially constructed. The transform and quantization tasks, as well as the dequantization and inverse transform, are presented together, due to the low computational requirements and simpler parallelization model. When the measured execution time of each module is considered its distribution parameter, a weighted Directed Acyclic Graph (DAG) is obtained. The several nodes of such a graph (A, B,..., H) are the decision points, where each task can be submitted to any of the two processing devices. The edges represent the individual task transitions, weighted by the respective computing and data transfer times. The shortest path between the starting and ending nodes of this graph represents the minimum encoding time. Dijkstra's algorithm [18] is typically used to find such a path, by defining, for each encoding module of the *inter*-loop, the processing device on which it will be executed when encoding the subsequent frames. Due to the reduced number of nodes and edges, the application of this algorithm does not add a significant delay to the encoding procedure.

4 Parallel video coding algorithms for GPU and CPU platforms

Even though the proposed task distribution and computational load balancing scheme can significantly shorten the processing time, the encoder overall performance still heavily depends on the parallel algorithms that are used for each task on the CPU and on the GPU. The fine-grained parallelization approach that is adopted in the GPU demands the development of highly efficient parallel

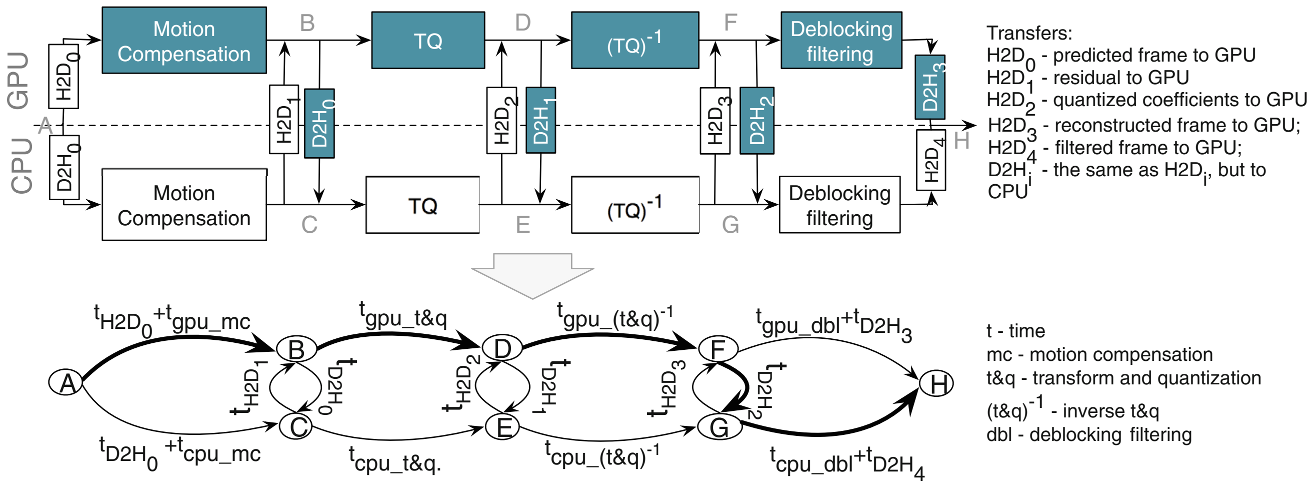

GPU
CPU
Motion Compensation
TQ
(TQ)^-1
Deblocking filtering
A B C D E F G H
H2D0 D2H0 H2D1 D2H0 H2D2 D2H1 H2D3 D2H2 D2H3 H2D4
Transfers:
H2D0 - predicted frame to GPU
H2D1 - residual to GPU
H2D2 - quantized coefficients to GPU
H2D3 - reconstructed frame to GPU;
H2D4 - filtered frame to GPU;
D2Hi - the same as H2Di, but to CPU
tH2D0+tgpu_mc
tgpu_t&q
tgpu_(t&q)^-1
tgpu_dbl+tD2H3
tD2H0+tcpu_mc
tcpu_t&q.
tcpu_(t&q)^-1
tcpu_dbl+tD2H4
tH2D1 tD2H0 tH2D2 tD2H1 tH2D3 tD2H2
t - time
mc - motion compensation
t&q - transform and quantization
(t&q)^-1 - inverse t&q
dbl - deblocking filtering

Fig. 6 Construction of weighted DAG from the data-flow diagram of the H.264/AVC encoder and a possible minimal path (represented in bold)

algorithms, able to concurrently process a large number of data independent thread blocks, and provide highly optimized parallelization on thread-level [16, 17]. On the other hand, the CPU parallelization comprises not only extensive core-level parallelization (e.g., by using the OpenMP API [19]), but also rather optimized Single Instruction Multiple Data (SIMD) vectorization (e.g., based on the SSE [20] instruction set). More detailed explanation of these algorithms can be found in [21].

4.1 GPU parallel algorithms

In most of the parallel algorithms developed for the GPU, each thread-block performs the computations related to a MB, and the set of MBs of a entire frame under processing are examined (at once) in a grid. The ME, interpolation and SME parallel algorithms for the GPU are based on the algorithms proposed in [5]. The *full-pixel ME* algorithm consists of three sub-steps, namely: *i)* caching of the MB and the search area pixels; *ii)* the analysis of the prediction candidates; and *iii)* finding of the best matching candidates. While the MB caching is performed only once, in such a way that each thread loads a single pixel, the search area needs to be cached for each RF. In the case of very large search areas, the corresponding pixels are cached and examined in several sub-areas, chosen in such a way that they individually fit to the local caches. An exhaustive Full-Search Block-Matching (FSBM) algorithm with a re-usage optimization of the partial SAD values (FSBMr) is adopted for the ME module, to eliminate data-dependencies between the thread-blocks and to exploit a hierarchical SAD computation scheme. The motion vector costs are added to the SAD values to obtain the distortion values [1]. Each thread within a 16×16 thread-block examines a subset of the best-matching candidates, keeping those with minimum distortion for all the sub-blocks in local registers. In order to obtain, in a single instruction, the minimum of two distortion values together with the related motion vectors (MVs), their values were concatenated in pairs ($distortion|MV$). In practice, the distortion value is shifted left 16 (bits) positions, and the motion vector is added to the same register. Those candidates are afterward compared with each other, to obtain the final predictor. An optimized reduction process for finding the minimum distortion candidates for all the sub-blocks [5] was adopted.

The *interpolation* module is implemented in such a way that each thread-block interpolates the pixels corresponding to a single MB, where each thread computes the set of sub-pixels around a pixel. Considering the fact that a different granularity of the MB-partitions (sub-blocks) implies a different number of examined candidates (implying a different parallelization potential of the related algorithms), the *sub-pixel ME* module is implemented by using 7 distinct kernels, each one for a single partitioning mode. These kernels are data-independent and can run in concurrent CUDA-threads. Due to the fact that for each sub-block the search algorithm starts from a different position (found with the full-pixel ME procedure) and considers a different search area, the search area pixels cannot be reused in the general case. Consequently, they have to be accessed directly from the GPU DRAM. In the sub-pixel refinement process, 25 best matching candidates are examined for each MB partition. In the proposed approach, the size of the thread block is $25 \times n$, where n is the number of the MB partitions in each mode. As an example, for the 4×4 mode there will be 25×16 threads in a thread block.

For the *direct and inverse integer transform* modules, each thread-block processes a single MB by using 16×4 threads. When the transform is performed in the vertical direction, each thread computes four coefficients in a single column, while in the horizontal direction each individual thread computes one coefficient within each of four adjacent columns. The *quantization and dequantization* modules are also performed within the same kernel, to reuse the already computed offset addresses for the pixel memory. However, all these threads must be synchronized and their outputs are written in different output buffers.

In the *deblocking filtering* module, each MB cannot be filtered until its left and upper neighbors have been processed. Consequently, the maximum number of MBs that are filtered in parallel is equal to the number of MBs within each anti-diagonal of the frame/slice under processing. This pattern seriously limits the attained performance, since the number of thread-blocks should not be less than the number of multi-processors in the GPU, to achieve the maximum efficiency. Within each thread-block, it was applied the same processing strategy that was adopted for the integer transform module (16×4 thread block size). In the presented optimized implementation, each thread filters one pixel of the vertical edge. Considering four 16 pixels vertical edges in each MB, only 64 threads operate in a single thread block. Since the filtered vertical edges are subsequently used for the processing of the horizontal edges, they are written back to the GPU cache in a transposed order, to minimize the bank conflicts. The algorithm is repeated as many times as the number of MBs within a column (wavefront model [22]). Despite all the considered optimizations, this algorithm cannot avoid branch divergence, but it takes advantage of adaptivity to reduce the computation.

4.2 CPU parallel algorithms

The implementation of the several encoder modules on the CPU also exploits the thread-level parallelism, where each core runs a single thread and processes a subset of the MB-rows. Besides this thread-level parallelization approach, the implemented algorithms also exploited Single Instruction Multiple Data (SIMD) vectorization, by using the SSE4 extension of the instruction set, where multiple pixels are loaded into vector registers and the same operation is simultaneously applied on all of them [20].

For CPU *full-pixel ME*, the parallelization at the thread level is applied regarding different MBs rows. The vectorization, however, is a major challenge. The main aspects that need to be taken into account are the pixel transfers into the XMM vector registers and the SAD computation. While the MB needs to be loaded once into XMM registers, the pixels of the best matching candidates need to be exchanged. Hence, to prevent an eventual performance drop, the candidates are examined in a column major order, instead of the usual row major. In such a way, for every next candidate in the same column, only one vector needs to be updated, while the rest of them are reused.

The SAD computation is based on the SSE4 MPSADBW instruction, which operates on 4-byte wide chunks and produces eight 16-bit SAD results. Each 16-bit SAD result is formed from overlapping pairs of 4-byte fields in the destination with the 4-byte field from the source operand. These destination fields start from 8 consecutive positions. In such a way, it is possible to compute eight 4×4 SAD with only 4 instructions, giving an average of 0.5 instructions per 4×4 SAD. The hierarchical computation of the SAD values is performed by applying the vector addition operation, as well as computation of the distortion values (by adding the motion vector cost to the SAD value). The obtained $distortion|MV$ pairs are packed in the same way as in the case of GPU implementation. This approach significantly decrease the number of clock cycles to obtain the minimum distortion motion vector, due to the fact that only the vector instructions are used for the comparison of the distortion values and the updating of the minimum distortion value and the motion vector, and no conditional branches are required.

Just as for the ME, the *interpolation* and the *SME* on the CPU side is computed using both thread-level parallelism at the level of MB-rows and logical and

Table 1 Hybrid (CPU + GPU) platforms adopted in the considered evaluation

	Platform 1		Platform 2		Platform 3	
	CPU	GPU	CPU	GPU	CPU	GPU
Model	Intel Core i7	GeForce 580GTX	Intel Core i7	GeForce 285GTX	Intel Core 2 Quad	GeForce 580GTX
# Cores	4	512	4	240	4	512
Frequency	3 GHz	1.54 GHz	3 GHz	1.48 GHz	2 GHz	1.59 GHz
Memory	4 GB	1.5 GB	4 GB	1 GB	4 GB	1.5 GB

Table 2 Motion estimation time per frame (ms) before and after load balancing, when considering 3 RFs

	720×576		1280×720		1920×1080	
	CPU	GPU	CPU	GPU	CPU	GPU
CASE 0: independent execution in both platforms—no distribution:						
ME	24.67	8.1	56.28	27.16	113.28	61.03
Interpolation	1.06	0.7	2.54	1.5	5.71	3.65
Task (total)	**25.73**	8.8	**58.82**	28.66	**118.7**	64.68
CASE 1: hybrid—half–half distribution:						
n_{CPU}/n_{GPU}	22	23	40	40	60	60
ME	12.3	4.1	27.79	13.58	56.2	30.4
Interpolation	–	0.7	–	1.51	–	3.65
Task (total)	**12.3**	4.8	**27.79**	15.09	**56.2**	34.05
CASE 2: hybrid—balanced distribution:						
s_{CPU}/s_{GPU}	1.82	5.55	1.42	2.95	1.05	1.96
n_{CPU}/n_{GPU}	12	33	27	53	45	75
ME	6.58	5.94	19.1	17.9	42.08	38.14
Interpolation	–	0.7	–	1.51	–	3.65
Task (total)	6.58	**6.64**	19.1	**19.41**	**42.08**	41.79

arithmetical vector instructions. However, in the particular case of the *interpolation* process, due to the fact that 1-byte pixels are used, the 6-tap filtering cannot be directly applied due to overflow of the addition and shift-left operations. Therefore, the PMOVZXBW vector instruction is initially applied, to extend the pixels to 2-bytes. The interpolated frame must be also conveniently prepared for vectorized SME. Namely, the best matching candidate pixels used for the SAD calculation with sub-pixel resolution are separated by 1 full-pixel from each other, which means 4 quarter-pixels apart. However, the SAD vector instruction can be only applied on the pixels placed in successive memory positions. Therefore, instead of storing each interpolated frame in a single 2D array, it is stored in 4 *interpolated subframes*, where the sub-pixels with the same distance from the full-pixel position are stored in the same sub-frame.

Contrary to what happens with the full-pixel ME, in the case of the *sub-pixel ME* the MPSADBW instruction is not perfectly suited for SAD calculation, since the consecutive candidates are not adjacent in memory. The PSADBW is used instead. The PSADBW is an SSE2 instruction which works on eight 2-byte chunks, being capable of processing 4 pixels wide chunks. However, since the XMM registers are 16 bytes wide, it is possible to examine two neighboring MB partitions in parallel, giving an average of 2 instructions per 4 × 4 SAD. The rest of the algorithm is the same as for the case of the full-pixel ME.

The *direct and inverse integer transform* modules are performed in separate vertical and horizontal steps. After the completion of each of them, the transpose operation over the 4 × 4 sub-blocks has to be performed. Both the transform and the transpose operate on two 4 × 4 sub-blocks at once. In the case of the *(de)quantization* modules, the 16 consequent pixels are concurrently (de)quantized, by mainly applying the vector shift operations.

Due to the high adaptivity of the *deblocking filtering*, the vectorization of this module represents the most difficult challenge on the CPU side. Just as in the GPU, the deblocking is performed in two steps, by filtering the vertical and the horizontal edges. In the first step, the loaded vectors need to be transposed, to apply the vector instructions. Apart from transposing the MB, the three transposed vectors from the left neighbor also need to be computed, to filter the left-most edge. Due to the 16 byte size of the XMM registers and the 2-byte pixels' format, the vertical filtering is done in 4 sub-steps, one for each 8 × 8 MB partition. Prior to each edge filtering step, the boundary strength is checked to verify if any filtering needs to be done. The procedure for the horizontal filtering is entirely similar, with the difference that the transpose operations are not needed.

Although the edge filtering is the same for both cases, its high adaptivity causes that different operations need to be applied for different edge pixel values. Consequently, the vectorization can only be done if all the possible branches

Table 3 Fine tuning of task distribution for the sub-pixel refinement module: execution time (ms) per frame for different configurations

	Task/ platform	720 × 576		1280 × 720		1920 × 1080	
		GPU	CPU	GPU	CPU	GPU	CPU
	Profiling phase						
Iteration 0:	sub_4 × 4	0.45	3.31	0.99	8.10	1.44	18.45
	sub_8 × 4	0.53	1.53	0.91	3.81	1.35	8.57
	sub_4 × 8	0.38	2.86	0.80	6.96	1.38	15.66
	sub_8 × 8	0.48	1.20	1.03	3.16	1.51	7.11
	sub_16 × 8	0.50	0.84	1.07	2.17	1.59	4.81
	sub_8 × 16	0.44	0.93	0.94	2.36	1.57	5.31
	sub_16 × 16	0.60	0.73	1.32	0.86	2.04	1.89

	Subtask on CPU	GPU	CPU+Data Transf.	Total	GPU	CPU+Data Transf.	Total	GPU	CPU+Data Transf.	Total
	Distribution phase									
Iteration 1:	none	**3.09**	–	3.09	**6.87**	–	6.87	**10.88**	–	10.88
Iteration 2:	16 × 16	**2.52**	0.73 + 0.01	2.52	**5.55**	0.86 + 0.01	5.55	**8.84**	0.02 + 1.89	8.84
Iteration 3:	prev. + 16 × 8	**2.05**	1.58 + 0.01	2.05	**4.53**	3.01 + 0.02	4.53	**7.25**	6.71 + 0.02	7.25
Iteration 4:	prev+8 × 16	1.64	**2.51 + 0.02**	2.53	3.63	**5.39 + 0.03**	5.42	5.68	**12.04 + 0.03**	12.07
	Steady-state load balancing									
		1.92	1.91	**1.92**	4.11	4.12	**4.12**	7.13	7.1	**7.13**

are executed first and the correct ones are selected afterwards, according to the observed branch conditions. For that purpose, the PBLENDVB vector instruction is used; this instruction selects the elements from one of two input vectors, according to a predefined pattern generated by the comparison instructions.

5 Experimental results and evaluation

The validation of the proposed task and load distribution model was conducted with a H.264/MPEG-4 AVC encoder, based on JM 18.4 reference software [23]. The considered test video sequences were *crowd_run*, *park_joy*, *blue_sky*, and *rush_hour*, with resolutions of 720 × 576, 1280 × 720 and 1920 × 1080 pixels, using a search range of 32 × 32 pixels for the ME module. The *IPPP* prediction structure was considered, together with the baseline H.264/AVC profile. The video coding parameters were chosen according to recommendation [24].

The used platforms to test and evaluate the proposed methods and parallelization algorithms are presented in Table 1. Moreover, all these platforms used Linux operating system, CUDA 4.1 framework, icc 12.0 compiler, and OpenMP 3.0 API. As it can be seen, *Platform* 2 has a computationally less powerful GPU and the same CPU as *Platform* 1, while *Platform* 3 has a slightly faster GPU and a significantly less powerful CPU. In the following sections, *Platform* 1 will be adopted for the main experimental procedures, unless otherwise stated.

5.1 Load balancing of the full-pixel motion estimation and interpolation modules

Due to the involved computational cost, an efficient load balancing of the ME task is fundamental for the overall performance of the encoder. The obtained results for the full-pixel ME module with load balancing (CASE 2) are presented in Table 2 and compared with the two simpler approaches: without any distribution (CASE 0) and when a straightforward distribution is used, considering that one half of the frame is sent to each of the two processing devices (CASE 1). The processing time for the completion of the ME module (corresponding to the last finished device) for each resolution is represented in bold in the *Task (Total)* rows of this table. The experimental results were obtained using *Platform* 1.

From the obtained results, it can be observed that the processing time is significantly reduced even with a simple offloading of the ME task for half of the frame (see execution times corresponding to CASE 0 and CASE 1). However, the observed execution times in the two devices still significantly differ. In fact, only with the application of the proposed method (CASE 2) it was possible to achieve a balanced execution time in the CPU and in the GPU, corresponding to the fastest processing time and to a significant speedup, when compared to CASE 1. Although not significant, the difference of the observed processing times in the CPU and in the GPU is mainly due to the considered chunk granularity, corresponding to a division by MB-columns. In fact, this balance could still be improved, by further refining this granulation.

In fact, according to the load balancing method presented in Sect. 3.2, the frame was divided considering not only the predicted performance of the ME and the interpolation modules, but also the related data transfers. However, while the transfer time of the current frame is included in the evaluation of the ME performance on the GPU, the transfer of the interpolated frame is completely overlapped with the computational part of the ME module. In such a way, due to the ability of the GPUs to overlap the data transfers and the computations, the largest and the most time demanding transfer (the interpolated frame is 16 times larger than the initial frame) of the proposed *inter-*

Table 4 Processing time of the least time consuming modules [ms] (with 3 RFs) for the proposed method. H2D - host (CPU) to device (GPU) transfer; D2H—device (GPU) to host (CPU) transfer

Resolution	Device	Transfers		MC	Transfers		T&Q	Transfers		DBL	Transfers		Total
		H2D	D2H		H2D	D2H		H2D	D2H		H2D	D2H	
720 × 576	CPU			0.27			0.35			**0.55**			**1.11**
		0.01 ↓	0.02 ↑		0.12 ↓	0.15 ↑		0.12 ↓	**0.15** ↑		**0.12** ↓	0.15 ↑	
	GPU			**0.14**			**0.14**			1.61			
1280 × 720	CPU			**0.33**			**0.4**			**0.95**			**1.97**
		0.03 ↓	**0.05** ↑		0.2 ↓	0.27 ↑		0.24 ↓	0.27 ↑		**0.24** ↓	0.27 ↑	
	GPU			0.34			0.44			2.61			
1920 × 1080	CPU			0.63			1.09			**2.45**			**4.46**
		0.05 ↓	0.07 ↑		0.4 ↓	0.43 ↑		0.4 ↓	**0.43** ↑		**0.4** ↓	0.43 ↑	
	GPU			**0.56**			**0.57**			3.48			

loop algorithm is hidden behind the processing of the most computationally complex module. Consequently, the list of interpolated frames is kept updated on both devices, which allows both an efficient distribution of the SME, and the instantiation of the motion compensation on any of the available devices, since the interpolated frame is required as input of these modules. Following the transfer of the interpolated frame, the remaining part of the current frame is subsequently transferred in the same way, due to the fact that the processing time of the ME is significantly larger than the time required for these transfers. On the other hand, the transfer of the last RF is performed at the end of the *inter*-loop, thus keeping the list of the RFs updated on both devices (see Sect. 3.1).

5.2 Load balancing of the sub-pixel motion estimation module

The application of the proposed computational load distribution method to the SME module is presented in Table 3, regarding to three different resolutions. Iteration 0 represents a *profiling* iteration, where all the prediction modes are sequentially executed on both devices: GPU and CPU. While the execution time on the GPU side is not significantly different for the several different sub-tasks (sub-block modes), on the CPU side it is observed that the prediction modes with smaller sub-blocks perform significantly slower, especially for the blocks with the smallest width ($4 \times n$). This was already expected, since the finer granulation of the MBs leads to more partitions to be analyzed. On the other hand, for larger MB partitions, the 16 bytes wide vector registers (XMM) make the CPU more efficiently used when compared with the case of smaller partitions, which needed to be serially processed due to the spatially distant search areas (defined by the best motion vector found in the ME process). As a consequence, the computations not only need to be repeated as many times as the number of sub-blocks to be processed, but they also have to be separately (i.e. serially) implemented for each MB partition.

The load distribution effectively starts solely on the GPU, with a parallel execution of the kernels corresponding to all sub-tasks in different CUDA streams (iteration 1). Then, the 16×16 mode is the first to be offloaded to the CPU (iteration 2). Since the encoding time on the GPU is still larger, the process is continued. After offloading the 16×8 and the 8×16 modes, the CPU becomes slower and the offloading process is stopped (iteration 3 and iteration 4). With such configuration, corresponding to offloading these three SME modes to the CPU, the total encoding time is now significantly smaller. In fact, although the GPU now finishes slightly earlier than the CPU, both processing devices are now busy most of the time, thus greatly improving the efficiency of the parallel encoder implementation. It should be mentioned that rather small data-transfer times must be also taken into account to perform an asynchronous transfer of the interpolated frame (the largest transfer) with the execution of the full-pixel ME. Nevertheless, this transfer is hidden behind the computation and only the transfers corresponding to the computed motion-vectors impose some delay.

Hence, from the obtained experimental results it was observed that the application of the proposed method to the SME module provides a reduction of the execution time by at least 35 %. It was also shown that this method scales well with the frame resolution, by keeping a quite similar improvement rate. The experimental results were obtained using *Platform* 1.

5.3 Scheduling of the remaining modules

As soon as the most computationally demanding modules are distributed over the processing devices available on the hybrid platform, the several remaining modules of the H.264/AVC encoder need to be distributed. This distribution is performed dynamically, according to the previously achieved performance assessment, and without assuming any a priori allocation of the tasks. Table 4 presents the execution time per frame, obtained when applying the algorithm described in Sect. 3.4. Due to the reduced computational complexity, and to their partially synergetic implementation, the (de)quantization and the (inverse) integer transform algorithms were presented with a single node in the Dijkstra's algorithm.

The values presented in the top and in the bottom parts of each of the three rows of Table 4 represent the average

Table 5 Comparison of the *inter*-loop processing time per frame [ms] achieved for different parallelization methods on various platforms

Resolution	Platform 1			Platform 2			Platform 3		
	720 × 576	1280 × 720	1920 × 1080	720 × 576	1280 × 720	1920 × 1080	720 × 576	1280 × 720	1920 × 1080
CPU-only	38.3	87.72	184.76	38.3	87.72	184.76	50.11	117.38	264.78
GPU-only	14.07	38.72	80.17	46.25	102.43	256.01	15.76	34.1	77.73
Chen_original	22.72	50.57	105.21	52.95	119.14	298.52	22.54	50.42	123.78
Chen_optimized	13.35	37.4	79.73	49.39	106.08	268.02	17.48	44.2	104.24
Proposed	9.54	25.2	52.66	18.8	40.12	98.02	12.25	24.98	57.01

execution time per frame (in ms) for the CPU and GPU algorithms, respectively, considering the modules that are assigned using the Dijkstra's algorithm. On the other hand, the values in the middle of each row represent the average time per frame that is required for the data transfers between the CPU and the GPU (and vice versa), whenever some exchange of the processing device is required for the H.264/AVC modules under processing. The modules and the data transfers are represented by using their execution order, following the right hand direction. The bold faced values represent the modules and the data transfers chosen by Dijkstra's algorithm to minimize the overall processing time (similar to Fig. 6, Sect. 3.4). The initial data transfers (prior to the *MC* module) are related to the data transfers of the results obtained from the SME module, regarding the distortion values and the displacement to the best matching candidates. The transfers before and after the *T&Q* module are related to the residual signal and to the reconstructed frame, respectively. These values are equal, since they represent the transfer of a single frame. Finally, the data transfers after the *DBL* module are related to the RF pixels that are needed on both devices prior to the processing of the following frame. These experimental results were obtained using *Platform* 1.

As it can be observed from the values represented in bold face, a direct application of Dijkstra's algorithm suggests different paths for different resolutions. In particular, a CPU-only configuration (where the data transfers between the devices are not required) is suggested for the 1280 × 720 resolution. For the remaining two resolutions, the motion compensation (MC) and the transform and (de)quantization (T&Q) modules (including their inverses) are sent to the GPU. As soon as the T&Q module finishes the processing, the reconstructed frame is sent to the CPU (*D2H*), where the deblocking filtering is performed. At the end of each of the presented solutions, the RF is sent to the GPU, to keep the lists of the RFs updated on both processing devices.

Even though the achieved gain (in terms of the processing time) seems to be not particularly significant when compared

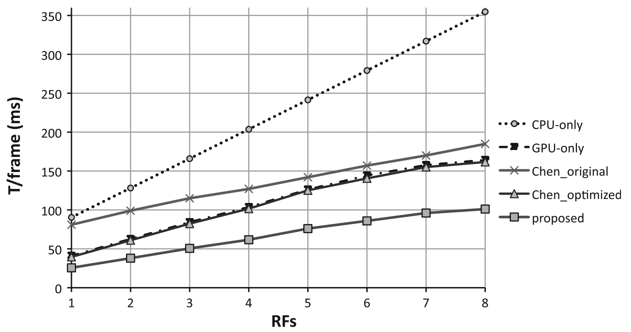


Fig. 7 Encoding time per frame [ms] when a distinct number of RFs is considered using the 1920 × 1080 video format. Comparison of the proposed approach with the CPU-only, GPU-only, and the original and optimized approaches proposed by Chen and Hang [12]

to the overall performance of the video encoder, it is recommended to perform this automated distribution instead of a direct submission of the tasks to the devices, according to any pre-defined path. In fact, the execution times of such modules highly depend on various system parameters, such as the offered computational power of the devices, the resolution of the video sequence, etc., which makes the optimal path to vary widely with these parameters.

5.4 Overall performance evaluation

The overall processing time of the whole *inter* loop of the video encoder is presented in Table 5 for the three hybrid platforms presented in Table 1, considering 3 RFs. The presented table compares the resulting performance (in encoding time per frame, ms) of five different scheduling strategies: *CPU-only*: the whole encoder is implemented in the CPU; *GPU-only*: the whole encoder is implemented in the GPU; *Chen_original*: method proposed by Chen [12], where the ME, SME and interpolation modules are implemented as in Sect. 4 and are statically offloaded to the GPU (the rest are kept in the CPU); *Chen_optimized*: Chen's encoder [12], optimized with OpenMP and SSE4 vectorization techniques, as presented in Sect. 4; *Proposed*: proposed dynamic load distribution strategy.

Contrasting to Chen's approach [12], which considers a static offloading to the GPU of only the ME, SME and interpolation modules, the proposed distribution method combines an adaptive data-level partitioning (see Eq. 5) and a dynamic selection of the device that offers the best performance for each of the processing modules of the video encoder (see Fig. 6).

The ME module of all these implementations (except the *CPU-only*) adopted the improved search algorithm proposed in [5], which proved to have superior performance when compared with the original Chen's algorithm. Furthermore, OpenMP parallelization techniques to exploit the available number of CPU cores were extensively considered (except in *GPU-only* and *Chen_original*), as well as a broad set of CUDA optimizations to exploit, as much as possible, the GPU computational resources (except in *CPU-only*).

In the case of Platform 1, the *GPU-only* configuration achieves more than 2 times higher performance when compared with the *CPU-only*. Nevertheless, the *Proposed* model achieves a much greater performance, corresponding to a speedup of up to 1.5 when compared with both *GPU-only* and *Chen_optimized* distribution.

As it was already expected, in the case of Platform 2 the GPU device is much slower and achieves a lower performance level than the CPU equipping this platform. Consequently, the implementation with *Chen_optimized* method is slower than the *CPU-only* configuration. Even so, the *Proposed* method using this slower GPU still achieves a speedup of at least 2.5 when compared with the other implementations, due to a selective distribution of the several modules that made it possible to efficiently exploit the CPU computational power in parallel with the GPU. The impact of the considered OpenMP and SSE4 vectorization [20] optimizations of the CPU code is emphasized in the presented results, by comparing the encoding times obtained for the *Chen_original* and *Chen_optimized* approaches.

Finally, Platform 3 represents a hybrid platform characterized by a higher GPU performance and a lower CPU performance. Therefore, the implementation with *Chen_optimized* method is significantly slower than the GPU-only implementation. However, the *Proposed* method was still able to improve the offered performance, achieving a speedup of about 1.4 when compared with the *GPU-only* implementation.

From the obtained results, it can be observed that the offered speedup generally grows with the adopted spatial resolution of the processed video sequences. Furthermore, contrasting with *Chen_optimized* and *GPU-only* approaches, the *Proposed* algorithm achieves a higher performance that is less dependent on the characteristics of the adopted hybrid platform. In particular, by simultaneously using the computational resources of the CPU and the GPU devices, in an adaptive and dynamic load balanced fashion, both processing devices participate towards a much better distribution of the computational load, based on a constantly updated prediction of the offered performance by each device.

The chart illustrated in Fig. 7 presents the comparison of the resulting encoding time (per frame) when the proposed load distribution method is compared with the same algorithms that were considered in Table 5. The chart was obtained for distinct parameterizations in what concerns to the number of RFs. Platform 1 was used for this evaluation.

When compared with the *GPU-only* implementation, a speedup magnitude of 1.5 is achieved with the *Proposed* method. However, when compared with the simplest (and most often used) *CPU-only* implementation, the attained speedup grows with the number of RFs, achieving magnitudes between 3 and 4. Moreover, it is worth to mention that the optimized implementation of the technique proposed by Chen and Hang [12] (*Chen_optimized*) only achieves a slightly better performance than the *GPU-only* implementation, mainly due to the faster deblocking filtering in the CPU device. However, the advantage of Chen's model, when compared with the *GPU-only* implementation, is the fact that it does not need to implement all the algorithms in the GPU. On the other hand, the clear improvement that is achieved with the *Proposed* model is mainly obtained with sub-task division and distribution of the most computational demanding modules (mainly, the ME and the SME modules) among the two processing devices, while keeping the rest of the modules in the fastest device (usually, the GPU).

Finally, contrasting with the other considered techniques, it is worth mentioning that by applying the proposed load balance and task distribution method a real-time video encoding platform was achieved for HD video sequences of 1920×1080 pixels, with a frame rate corresponding to almost 40 frames per second when using a single RF. It is worth noting that the proposed lightweight scheduling technique, together with the corresponding load-balancing scheme, take less than 1 *ms* to be executed on all the tested platforms.

6 Conclusions

A dynamic load balancing and task distribution model for AVC implementations on hybrid CPU + GPU platforms is proposed in this manuscript. The presented scheme performs the distribution of the computational load both on the inter-module and the intra-module levels, by extensively using all the resources that are offered by these devices for data parallelism. The possibility of asynchronously processing on the CPU and on the GPU is effectively exploited to efficiently distribute the computational load among these processing devices. Such load balancing is performed by a dynamic performance prediction scheme, based on the previously measured processing times. The highly optimized implementation includes the whole *inter*-loop of the encoder on both platforms.

Based on the proposed method, it was achieved a speedup of the total *inter*-loop encoding time as high as 4, when compared with the usual CPU-only approach. A speedup of up to 2.5 was also observed when comparing with the GPU-only and a state-of-the-art technique [12], based on a static offloading of the most computationally intensive modules (ME, interpolation, SME) to the GPU, keeping the rest of the modules on the CPU.

With the described scheme, it was achieved a real-time encoding performance (up to 40 frames per second) to process (HD) video sequences with a resolution of 1920×1080 pixels on an off-the-shelf desktop system, even when considering all the sub-block prediction modes and an exhaustive ME algorithm.

Acknowledgments This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project PEst-OE/EEI/LA0021/2013.

References

1. Ostermann, J., Bormans, J., List, P., Marpe, D., Narroschke, M., Pereira, F., Stockhammer, T., Wedi, T.: Video coding with H.264/AVC tools, performance, and complexity. IEEE Circuits Syst. Mag. **4**(1), 7–28 (2004)

2. Wiegand, T., Schwartz, H., Kossentini, F., Ulivan G., S.: Rate-constrained coder control and comparison of video coding standards. IEEE Trans. Circuits Syst. Video Technol. **13**(7), 668–703 (2003)
3. Lu, C.-T., Hang, H.-M.:Multiview encoder parallelized fast search realization on NVIDIA CUDA. In: Proc. Visual Communications and Image Processing (VCIP), IEEE, pp. 1–4 (2011)
4. Schwalb, M., Ewerth, R., Freisleben, B.: Fast motion estimation on graphics hardware for H.264 video encoding. IEEE Trans. Multimed. **11**(1), 1–10 (2009)
5. Momcilovic, S., Sousa, L.: Development and evaluation of scalable video motion estimators on GPU. In: Proc. Workshop on Signal Processing Systems (SIPS) (2009)
6. Kung, M.C., Au, O., Wong, P., Liu, C.-H.: Intra frame encoding using programmable graphics hardware. In: Proc. Pacific Rim Conference on Advances in Multimedia Information Processing (PCM), pp. 609–618. Springer, Berlin (2007)
7. Obukhov, A., Kharlamovl, A.: Discrete cosine transform for 8x8 blocks with CUDA. Research report, NVIDIA, Santa Clara, CA (2008)
8. Shen, G., Gao, G.-P., Li, S., Shum, H.-Y., Zhang, Y.-Q.: Accelerate video decoding with generic GPU. IEEE Trans. Circuits Syst. Video Technol. **15**(5), 685–693 (2005)
9. Pieters, B., Hollemeersch, C.-F., De Cock, J., Lambert, P., De Neve, W., Vande Walle, R.: Parallel deblocking filtering in MPEG-4 AVC/H.264 on massively parallel architectures. IEEE Trans. Circuits Syst. Video Technol. **21**(1), 96–100 (2011)
10. Cheung, N.-M., Fan, X., Au O., C., Kung, M.-C.: Video coding on multicore graphics processors. IEEE Signal Process. Mag. **27**(2), 79–89 (2010)
11. Azevedo, A., Juurlink, B., Meenderinck, C., Terechko, A., Hoogerbrugge, J., Alvarez, M., Ramirez, A., Valero, M.: A highly scalable parallel implementation of H.264. In: Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC), pp. 111–134 (2011)
12. Chen, W.-N., Hang, H.-M.: H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA). In: Proc. International Conference on Multimedia and Expo (ICME), pp. 697–700 (2008)
13. Momcilovic, S., Roma, N., Sousa, L.: Multi-level parallelization of advanced video coding on hybrid CPU/GPU platform. In: Proceedings of the 10th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar/Euro-Par 2012) (2012)
14. Ates, H.F., Altunbasak, Y.: SAD reuse in hierarchical motion estimation for the H.264 encoder. Proc. IEEE Int. Conf. Acoust. Speech Signal Process. (ICASSP) **2**, 905–908 (2005)
15. First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem). Intel Corporation (2008)
16. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with CUDA. IEEE Micro **28**(4), 13–27 (2008)
17. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)
18. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
19. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, Cambridge (2007)
20. Intel Corporation. SSE4 Programming Reference (2007). http://edc.intel.com/Link.aspx?id=1630
21. Momcilovic, S., Ilic, A., Roma, N., Sousa, L.: Advanced Video Coding on CPUs and GPUs: Parallelization and RD Analysis. Technical report (available online), INESC-ID (2013)
22. Aji, A.M., Feng, W., Blagojevic, F., Nikolopoulos, D.S.: Cell-SWat: modeling and scheduling wavefront computations on the cell broadband engine. In: CF '08: Proceedings of the 5th Conference on Computing Frontiers, pp. 13–22. ACM, New York (2008) (ISBN 978-1-60558-077-7)
23. ITU-T. JVT Reference Software, version 17.2 (2010). http://iphome.hhi.de/suehring/tml/download
24. Tan, T.; Sullivan,G.; Wedi. Recommended simulation common conditions for coding efficiency experiments-revision 3. Doc. VCEG-AI10, ITU-Telecommunications Standardization Sector, STUDY GROUP 16 Question 6, Video Coding Experts Group (VCEG), Lisbon, Portugal (2008)

Author Biographies

Svetislav Momcilovic is a Senior researcher at Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID), working in the area of parallel video coding on multicore architectures. He received his Ph.D. in Electrical and Computer Engineering in 2011 from the Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal. His current research interests are mainly focused on parallel computing, multicore architectures, heterogeneous parallel systems and video coding, including high efficiency, advanced, multiview and distributed video coding. Svetislav Momcilovic is a member of IEEE.

Nuno Roma was born in Entroncamento, Portugal in 1975. He received the Ph.D. degree in electrical and computer engineering from Instituto Superior Técnico (IST), Universidade Técnica de Lisboa, Lisbon, Portugal, in 2008. He is currently an Assistant Professor with the Electrical and Computer Engineering Department, IST, and a Senior Researcher of the Signal Processing Systems Group (SiPS) of Instituto de Engenharia de Sistemas e Computadores R&D (INESC-ID). His research interests include specialised computer architectures for digital signal processing (including image and video coding/transcoding and biological sequences processing), embedded systems design and compressed-domain video processing algorithms. He has contributed to more than 50 papers to journals and international conferences. Currently, he is the principal investigator of a funded national project and he is member of several research Networks of Excellence (NoE), such as: "European Network of Excellence on High Performance and Embedded Architecture and Compilation" (HiPEAC), funded by the European Seventh Framework Programme (FP7); the "Open European Network for High Performance Computing on Complex Environments" (ComplexHPC - ICT COST Action IC0805); and the "Next Generation Sequencing Data Analysis Network" (SeqAhead - BMBS COST Action BM1006), funded by the European Cooperation in Science and Technology (COST) programme. Dr. Roma is a Senior Member of the IEEE Circuits and Systems Society and a member of ACM.

Leonel Sousa received the Ph.D. degree in Electrical and Computer Engineering from Instituto Superior Técnico (IST), Technical University of Lisbon, Portugal, in 1996. He is currently a Full Professor of the Electrical and Computer Engineering Department at IST and a senior researcher at INESC-ID. His research interests include VLSI architectures, parallel and distributed computing and multimedia systems. He has contributed to more than 200 papers in journals and international conferences. He is an associate editor of the Eurasip Journal on Embedded Systems and served in the program committee of several conferences. He is member of the Management Committee of EU COST Action ComplexHPC, member of HiPEAC/HiPEAC2 and HiPEAC Reconfigurable Computing Cluster. He is a Senior Member of IEEE (2004), Senior Member of ACM (2009), Member of IFIP WG10.3 on concurrent systems, and a fellow of the IET.