SIViP
DOI 10.1007/s11760-014-0633-8

ORIGINAL PAPER

Scalable row-based parallel H.264 decoder on embedded multicore processors

Elias Baaklini · Santhosh Rethinagiri · Hassan Sbeity · Smail Niar

Received: 29 April 2013 / Revised: 4 March 2014 / Accepted: 6 March 2014
© Springer-Verlag London 2014

Abstract Multimedia applications are present in most mobile hand-held devices, which are still equipped with limited battery resources. The H.264 standard is currently dominating the video compression world. H.264 has high computational requirements in terms of memory, energy, and time. Many techniques emerged that optimize parallel task granularity on multicore systems ranging from groups of pictures until the smallest block of pixels. A scalable parallel technique for the motion compensation phase is proposed in this research that is based on processing of groups of macroblock rows. Moreover, a light dependency detection algorithm is added to the prediction phase that enables parallel execution and minimizes synchronization stall time. Furthermore, a parallel implementation of the deblocking filter is also implemented. The overall result is an efficient and highly scalable parallel H.264 decoder that is evaluated on a real-board platform composed of an ARM Cortex-A9 MPCore with four processors. Various low- and high-definition video sequences are used in experiments. Results show that execution time reaches a speedup of 3.3× for motion compensation stage and an overall speedup of 2.3× on 4 cores including communication and synchronization overhead. Energy consumption decreases up to 63 % for the whole application execution.

E. Baaklini (✉) · S. Niar
University of Valenciennes, Valenciennes, France
e-mail: elias.baaklini@univ-valenciennes.fr

S. Niar
e-mail: smail.niar@univ-valenciennes.fr

S. Rethinagiri
Microsoft Research Center, Barcelona Supercomputing Center (BSC), Barcelona, Spain
e-mail: santhosh.rethinagiri@bsc.es

H. Sbeity
Arab Open University, Beirut, Lebanon
e-mail: hsbeity@aou.edu.lb

Keywords Multimedia · H.264/AVC standard · Video compression · Optimization · Parallel computing · Embedded systems · Multicore processors

1 Introduction

Mobile devices supporting multimedia applications are nowadays considered pervasive in our modern world societies. Smartphones and tablet devices are equipped with high-resolution screens and fast multicore embedded processors. Video players, digital cameras, televisions, and phones support high resolutions such as HD and Full-HD. However, few multimedia applications benefit from the computational potentials that multicore processors offer in these emerging powerful embedded devices. Furthermore, video coding standards such as H.264/AVC [8] and HEVC [20] are adopting complex algorithms like *context-adaptive binary arithmetic coding* (CABAC) and *in-loop deblocking filter* in order to achieve better compression and to lower transmission bitrates. However, the additional complexity of these algorithms has negative impacts on execution time and energy consumption.

H.264/AVC [8] is currently one of the most widely used video compression standards for recording, compressing, and distributing high-definition (HD) videos. The standard's first draft was released in 2003 and its latest version in 2012 [8]. Most HD video streaming websites like YouTube currently support H.264 as their default video codec [26]. H.264 is a high computational video compression standard that emerged as a result of the joint effort for Moving Picture Experts Group (MPEG) and the Video Coding Experts Group

(VCEG). The H.264 standard offers better compression and higher quality compared to other standards like MPEG-2 [22]. This increase in compression results is the cost of high computational blocks like *Deblocking Filters* (DF) and complex *Entropy Decoding* techniques (CABAC and CAVLC).

Nowadays, most system-on-chip (SoC) platforms have multicore processors. Dual and quad cores are found in recent smartphones and tablet devices like Samsung and Apple phones [1,16]. ARM Cortex-A9 processors can have up to 4 cores per chip [2]. Cortex-A15 processors can have up to 8 cores per chip (each chip can contain 2 clusters where each cluster can have up to 4 cores) [3]. On the other hand, applications do not benefit automatically from these powerful top-of-the-line processors. Even with new cutting-edge processors, video resolutions are increasing rapidly which require more processing time and consequently more energy consumption. Operating systems simply map independent applications, or multiple threads within an application, on different cores. Therefore, one application alone may not benefit from the additional resources available unless it is designed to execute in parallel. Thus, sequential applications need to be redesigned and recompiled in order to support parallelism. The process of parallelization faces many challenges such as dependencies, synchronization, and data coherency. In our research, we choose the H.264/AVC video decoder [8] as a high computational multimedia application to be parallelized. We solve the problem of high complexity of the H.264 decoder using parallel execution on multicore embedded processors.

Many parallel implementations exist ranging from parallel decoding of macroblocks (*fine-grain* implementations) till parallel decoding of groups of pictures (*coarse-grain* implementations). A macroblock is a 16 × 16 square pixel component of an image in a video sequence. Moreover, a macroblock can also be divided into subblocks of smaller size. Macroblock parallel decoding is highly scalable since many independent macroblocks can be processed in parallel. However, dependencies and huge overheads are created as a result of memory communication and execution synchronization between macroblocks. On the other hand, parallel decoding of groups of pictures requires large memory, especially for high-definition video sequences. In addition, they have a lower scalability than parallel macroblock decoding because of the small number of groups of frames that can be decoded in parallel. In our approach, we process rows of independent macroblocks in parallel using a new algorithm that eliminates dependencies between macroblocks and minimizes synchronization overhead. This level of parallel execution may be considered between the coarse-grain and the fine-grain parallel approaches, thus, offering a balance between large overheads and high scalability.

Our main contribution in this paper is the design and implementation of a new algorithm for processing macroblock rows of the H.264 decoder in parallel. In addition, a small footprint data dependency detection algorithm that isolates intra-prediction macroblocks (I-MBs) is implemented and executed on macroblocks of the same slice of a video frame. Experiments are conducted by executing our scalable parallel decoder on a Cuda Development Kit platform [13] with an ARM Cortex-A9 processor including 4 cores [2]. Execution time and energy consumption statistics are collected by running the application on the real-board platform. For HD and Full-HD resolutions, video sequences benchmarks reach their maximum throughput using 4 threads on 4 cores with a speedup of 3.3× for motion compensation and an overall speedup of 2.3× in terms of execution time and with an energy saving percentage of 63 %.

The remainder of the paper is organized as follows. In Sect. 2, we present the related work concerning H.264 parallel implementations. In Sect. 3, we describe background info related to H.264 decoding. In Sect. 4, we describe our approach for parallelizing the motion compensation phase and the deblocking filter. In Sect. 5, we present our real-board experimental results for execution time and energy consumption. We also discuss and analyze simulated executions and the theoretical scalability of our algorithm. Conclusion and future work are given in Sect. 6.

2 Related work

Ever since the H.264/AVC standard [8] was published in 2003, researchers started to solve the high complexity issue of the new standard mainly using parallelism. Several modifications were suggested for the H.264 encoders and decoders to improve the performance in terms of execution time and memory usage. Parallel decoding techniques of H.264 start from the highest level, which is the group of frames or pictures (GOP), coarse-grain level, till the lowest level which is the block inside a macroblock, fine-grain level.

Kannangara et al. [9] reduced the complexity of the H.264 decoder (19–65 %) by predicting the SKIP macroblocks using an estimation based on a Lagrangian rate-distortion cost function. Our experimental results show a better overall speedup (230 %) and a better parallel scalability relative to the number of cores in a multicore processor. Gurhanli et al. [6] suggested a parallel approach by decoding independent groups of frames on different cores. The speedup is conditioned with the modification of the encoder in order to omit the start-code scanner process. Any modification to the encoder will require the exclusion of previously encoded video sequences, which will need to be re-encoded in order to benefit from the proposed approach. In our parallel implementation, we only modify the decoder, which support all previously encoded video sequences. Nishihara et al. [12] proposed a load balancing mechanism among cores where

partitions sizes are adjusted at runtime. The authors also reduced the memory access contention based on execution time prediction. Among frame-level and MB-level parallelization, Zhao et al. [27] proposed a wavefront algorithm for processing independent macroblocks within the same frame and among different frames. This method for parallel processing of macroblocks does not equally distribute workload of different cores as the number of independent macroblocks varies with time. Mesa et al. [11] proposed a similar approach, the 2D-wave, which decodes independent macroblocks in parallel on different cores. A good scalability is proved for high resolutions. Moreover, an advanced parallel technique that is based on the 2D-wave algorithm, the dynamic 3D-wave approach, is proposed by Meenderinck et al. [10]. The dynamic 3D-wave algorithm, which combines spatial and temporal MB-level parallelism, uses a dynamic scheduler that assigns independent macroblocks to parallel threads. The dynamic scheduler minimizes the differences in workload on different threads, and thus, it optimizes the parallel execution of independent macroblocks on parallel threads. Chong et al. [4] added a preparsing stage in order to resolve control dependencies for macroblock-level parallelism. Vandertol et al. [24] mapped video sequences data over multiple processors providing better performance over functional parallel algorithms. The authors group macroblocks in a frame with minimal dependency between cores. Horowitz et al. [7] compared different H.264 implementations including FFmpeg [5] and the H.264 reference software JM [19]. The authors also analyzed the complexity of the H.264 decoder subsystems. Sihn et al. [18] proposed a multicore pipeline for the deblocking filter based on the group of pictures data level partitioning. He also suggested software memory throttling and fair load balancing techniques in order to improve multicore processors performance when several cores are used.

Among the literature that already exists for parallel deblocking filter, Wang et al. [25] partitions a slice into independent rectangles with arbitrary granularity. These independent regions are identified by examining the influence of vertical and horizontal lines of pixels. Parallel deblocking of these regions has good scalability, minimal synchronization overhead, and good cache utilization. However, a small number of pixels will have erroneous output without affecting the overall deblocking filter process with what they refer to as the Limited Error Propagation Effect. For an optimized deblocking filter, a speedup of 95 and 224 % is achieved on 2 and 4 cores, respectively. For an H.264 decoder, the overall speedups are 21 % on 2 cores and 34 % on 4 cores. Pieters et al. [15] proposed a macroblock partitioning algorithm that is based on a parallel version described by Wang et al. [25] with the avoidance of the Limited Error Propagation Effect. The proposed algorithm filters the pixels of macroblocks concurrently. The parallel technique is also tested on GPU platforms. The parallel implementation outperforms both CPU-based and GPU-based implementations by a factor up to 10.2 and 19.5, respectively.

In our research, we optimize the H.264 decoder knowing that our approach is also applicable to the H.264 encoder. We focus on improving the efficiency of the H.264 decoder using multicore processors. We decode groups of rows of macroblocks in parallel where each group is mapped to one core. Dependencies between macroblocks are avoided by decoding intra-prediction macroblocks sequentially at the end of the decoding stage. We prove that our approach has a better load balancing on multiple cores in addition to lower synchronization overhead than other approaches. With these advantages, we eventually reach higher theoretical and realistic speedups. We evaluate our approach on a real platform equipped with a quad core processor. Execution time and energy consumption statistics are gathered and analyzed. As of our knowledge, our results are more realistic compared with other work carried out in the same literature.

In the following section, we briefly describe the H.264 decoder process. We also discuss the decoder's decomposition and its parallel execution possibilities.

3 H.264 background

In this section, we provide brief background information about H.264 video coding standard. We also introduce our parallel decoding approach for the standard.

3.1 H.264 features and tools

The H.264/AVC standard was designed for high compression efficiency, reliability, and flexibility, so that it could support a wide variety of applications and different types of communication such as wired and wireless networks.

3.1.1 Layer structure

The H.264 standard consists of various features and coding tools that contribute to the high compression efficiency, flexibility, and robustness. To achieve the flexibility, the standard was designed to contain two layers:

1. The *Video Coding Layer* (VCL) represents the video encoding process and the coded bits.
2. The *Network Abstraction Layer* (NAL) handles the transportation of VCL data and other header information by encapsulating them in NAL units.

The separation of video coding and transportation into two layers ensures that the video coding layer provides an efficient and adaptable representation of video content.

3.1.2 Profiles and levels

Profiles are used to specify the tools and capabilities of the decoder that is needed to support different applications. Each profile is designed to have particular coding tools to support various coding requirements. The H.264/AVC standard originally specified the following three basic profiles:

1. **Baseline**: low-latency, low-complexity, error resilience and robustness. Applications: video conferencing.
2. **Main**: high compression efficiency. Applications: video storage and broadcasting
3. **Extended**: superset of the baseline profile with enhanced error resilience and video stream switching capabilities. Applications: internet video streaming.

Levels provide inter-operability between different decoder implementations. They are defined as performance limits for decoders supporting each profile. Performance limits generally apply to processor load, memory capabilities, and the maximum bit rates supported by a decoder.

3.1.3 Picture structure

The source video is coded as a stream of pictures. The smallest coding unit in a picture is a *Macroblock* (MB). A macroblock contains data belonging to a region of 16×16 *luma* samples, Y (brightness), along with the associated *chroma* component samples, Cr (red) and Cb (blue).

A picture consists of one or more slices. Each slice contains an integral number of macroblocks, which should be processed in raster scan order. H.264 has the following slice types:

- **I-Slices:** all the macroblocks in the slice are coded using intra-prediction (using data already coded within the same slice).
- **P-Slices:** contains inter-coded macroblocks using one reference picture and intra-coded macroblocks (Predictive).
- **B-Slices:** contains inter-coded macroblocks using two reference pictures as well as macroblock types in P-slices (Bi-predictive).
- **SP and SI-Slices:** Special types of slices, Switching Predictive (SP) and Switching Intra (SI), for efficient switching between different video streams, random access and error resilience.

The number of slices and the number of macroblocks in each slice are flexible. Therefore, the encoder can decide on an appropriate size depending on the coding requirements. Slices are processed independently of each other. The independent decoding of slices adds robustness against data loss because the rest of the picture is not affected.

3.2 H.264 decomposition

The H.264 decoder can be divided into five main functional phases: *Entropy Decoder* (ED), *De-Quantization* and *Inverse Transform* (IQT), *Motion Compensation* (MC) and *Intra-Prediction* (IP), and *Deblocking Filter* (DF). The H.264 decoder stages are illustrated in Fig. 1. The decoder process starts by entropy decoding the input bitstream. Then, de-quantization and inverse transformation are applied to the resulting data. Afterward, in every slice of a frame, macroblocks are processed in raster mode. Each macroblock is intra- or inter-predicted (motion compensation) using the reference frames. The deblocking filter is applied at the end in order to make the edges between macroblocks smooth and invisible to human vision. Figure 2 illustrates the average execution percentage of each main phase using the baseline and the main profiles. De-quantization and inverse transform phase can be grouped with the entropy decoder phase because they have a small footprint on overall execution. Both predictions phases, motion compensation and intra-prediction, are also merged together into one phase. Our parallel algorithm is applied to the prediction phase that ranges from 41 till 45 % of the overall decoding process. The entropy decoder with de-quantization and inverse transform is executed sequentially with a percentage ranging from 14 till 19 %. We use

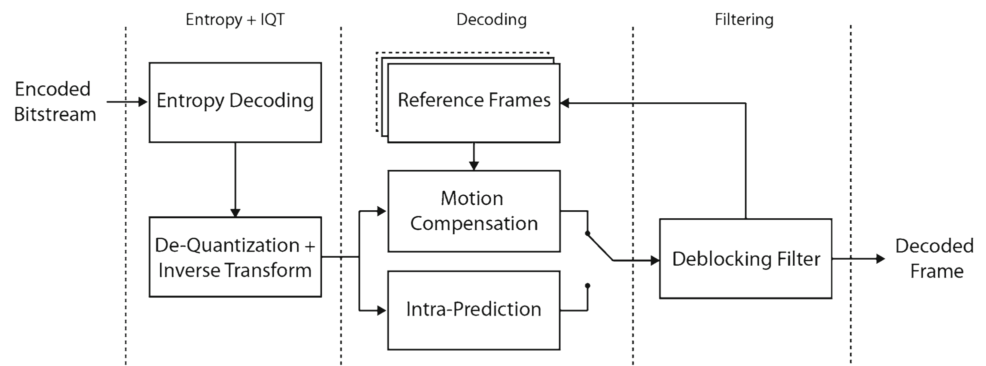

Fig. 1 H.264 decoding process

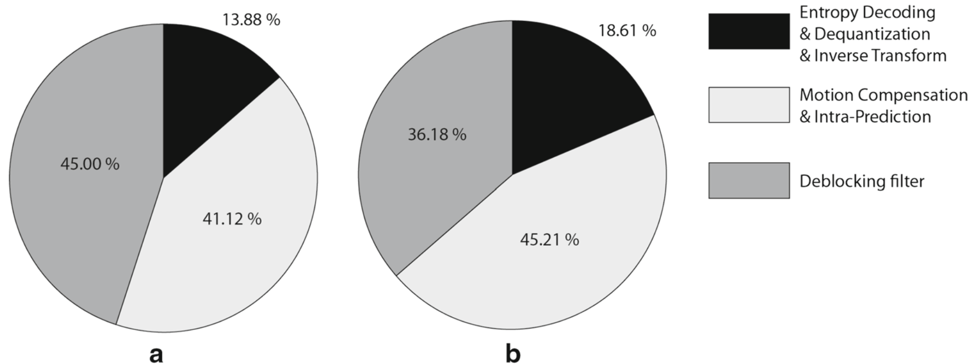

Fig. 2 H.264 decoding stages workload percentages. **a** Baseline profile. **b** Main profile

the wavefront algorithm [27] for the deblocking filter of the H.264 decoder. The deblocking filter has a huge impact on the overall performance of the decoder that is 45 % for the baseline profile and 36 % for the main profile.

3.3 H.264 macroblocks

Each slice of a picture frame is partitioned into square blocks of 16 × 16 pixels called *Macroblock* (MB). The number of horizontal and vertical macroblocks varies with the resolution of the frame. A macroblock can be divided into subblocks of 16 × 8, 8 × 8, 8 × 4, and 4 × 4 pixels. The encoder chooses the subblocks sizes depending on the amount of details (complexity) for specific parts of an image frame. An image, or part of an image, is considered complex when it contains objects with tiny details. For example, in a video of a flying bird with a consistent blue background, the encoder will divide the macroblocks in the region displaying the bird into subblocks smaller than 16 × 16, and the blue sky macroblocks will remain with the same of size of 16 × 16. The motion compensation stage uses a reference buffer in order to calculate the values of macroblocks in the current frame. The reference buffer contains a list of previously decoded frames. Macroblocks that are inter-predicted and motion compensated from previously decoded frames are either of type P or B (P-MBs and B-MBs). P-MBs depend on macroblocks in a previously decoded frame. B-MPs are calculated using macroblocks in two reference frames. Macroblocks that depend on other macroblocks in the current frame (called I-MBs) are intra-predicted. Finally, deblocking filtering is applied at the end of the decoding process in order to reduce the edging effect between macroblock borders.

In the following section, we describe in detail our parallel implementation of the H.264 decoder.

4 H.264 parallel implementation

In this section, we elaborate on our parallel implementation of the H.264 video decoder. We explain how we apply parallelism to the motion compensation and the deblocking filter stages of the decoder. We also discuss macroblocks partitioning and their dependencies.

4.1 Parallel motion compensation

The H.264 reference implementation, JM [19], is the reference implementation for the H.264 standard. In our research, we modified the JM [19] source code of the H.264 decoder in order to decode rows of macroblocks in parallel using the PThread library of the POSIX standards in C programming language.

A thread is created for every group of macroblock rows. Each thread is mapped to a core. The number of thread is specified by the user or the application. If the number of threads is greater than the number of cores, then the scheduler will assign more than one thread for one core. As shown in Fig. 3, each thread handles the motion compensation stage for a group of macroblocks rows. All threads should complete their task before moving on to the next phase, which is intra-predication for I-MBs.

The maximum number of parallel decoding blocks is equal to the number of macroblock rows. This level of parallel decoding of macroblock rows may be considered in

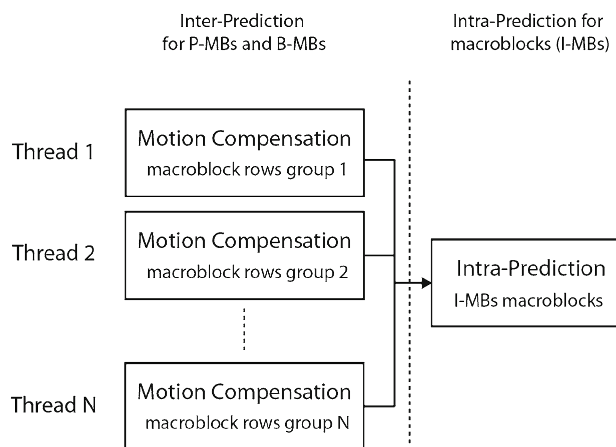

Fig. 3 Decoding groups of macroblock rows in parallel using N threads

between coarse-grain and fine-grain approaches. Coarse-grain approaches process multiple slices or frames in parallel. These high-level methods, like [6,9,12,18], need high memory usage in order to decode multiple frames in parallel because of the required size to store and to transfer data of several frames. Fine-grain approaches decode macroblocks or blocks inside a macroblock in parallel. These low-level methods, like [4,24,27], cause an enormous synchronization overhead affecting deeply the speedup for the reason of the large number of macroblocks in every frame. The balance between both approaches is also reflected on synchronization overheads and data communication requirements.

Our approach is aimed to benefit from the balance between both advantages and disadvantages. Macroblock rows require less memory than a frame and more than one macroblock. In fact, our approach is scalable up to the macroblock level. Such granularity will create a huge overhead of parallelism on current multicore architectures. On the other hand, the number of macroblock rows is much less than the total number of macroblocks. For example, in HD resolution (1,280 × 720), each frame has 3,600 macroblocks, 80 horizontal MBs, and 45 vertical MBs. Thus, the number of macroblocks rows is less by a factor of 80 than the total number of macroblocks. As a result, the overhead for synchronization and communications between cores is also reduced by a factor of 80.

4.2 Dependencies between macroblocks

In H.264, there are 4 types of macroblocks: I, P, B, and SKIP. Figure 4 illustrates the dependencies between macroblocks of types I and P. I-MBs depend on other macroblocks in the same slice of a frame as shown in Fig. 4-a where the macroblock pointed at by the arrows may be dependent on one or more macroblocks. P-MBs depend on macroblocks from previously decoded frames as shown in Fig. 4-b where the origin of the arrow is a macroblock in a previously decoded frame. Motion vectors info is required for P-MBs in order to reconstruct the coded macroblocks. B-MBs depend on past and future reference frames. They are available in B-Frames, and they can have one or two motion vectors. The SKIP macroblock data remain the same when it is compared with another macroblock in a previously decoded frame. So the motion vector differences are zero, and therefore, the prediction macroblock is simply copied as the reconstructed macroblock.

In a frame, all macroblocks can be processed in parallel except I-MBs because they depend on macroblocks, which are being decoded in the same slice. So a dependency identification procedure is needed to satisfy intra-prediction dependencies. In order to overcome this constraint, we start by decoding all macroblocks of type P, B, and SKIP in parallel. During this step, we skip all I-MBs and we save a reference to the skipped macroblocks for future processing. When this stage is completed, the remaining I-MBs macroblocks in the current slice are decoded sequentially as illustrated in Fig. 3. Among the remaining I-MBs, independent macroblocks can be processed in parallel as they depend on macroblocks in the same slice that are already processed. For simplicity and because of their small number in each frame (except I-Frames), we process I-MBs sequentially in our algorithm.

With this ordering mechanism, dependencies between macroblocks in the same slice are satisfied. Table 1 lists the percentages of I-MBs, P-MBs, and SKIP-MBs in the video sequences that we use in our experiments. The average number of I-MBs for all video sequences is about 2 %. I-MBs also exist in P-frames and B-Frames. The number of I-MBs in a P-Frame or a B-Frame depends on objects with high detailed and on objects rate of movements in the video sequences. P-Frames and B-Frames are mostly composed of P-MBs and SKIP-MBs with a small number of I-MBs. So the small number of I-MBs in P-Frames and B-Frames does not significantly affect the overall speedup for the parallel decoding of macroblocks.

4.3 IDR frame frequency

An encoded video always starts with an I-Frame (IDR), which is composed completely for I-MBs. This type of

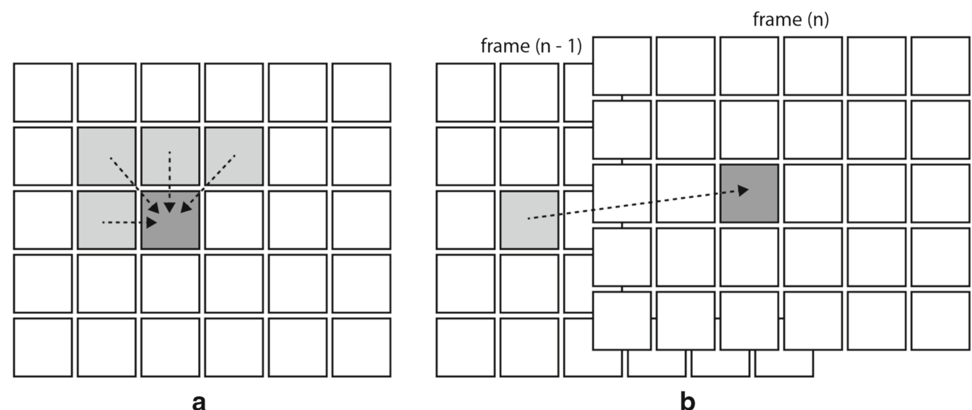

Fig. 4 Dependencies between macroblocks. **a** Intra-prediction. **b** Inter-prediction

Table 1 Percentages of different types of macroblocks per video sequence

Name	Resol.	Fr.	I	P	SKIP
Bus	352 × 288	150	1.70	79.20	19.10
Foreman	352 × 288	300	1.80	70.95	27.25
Waterfall	352 × 288	260	0.25	70.05	29.70
Johnny	854 × 480	600	0.10	22.35	77.55
Basketball	854 × 480	500	3.40	62.25	34.35
Cactus	854 × 480	500	1.50	42.30	56.20
Johnny	1,280 × 720	600	0.15	22.50	77.35
Basketball	1,280 × 720	500	3.95	58.50	37.55
Cactus	1,280 × 720	500	1.90	42.50	55.60
Basketball	1,920 × 1,088	500	4.95	55.30	39.75
Cactus	1,920 × 1,088	500	3.15	44.05	52.80
Terrace	1,920 × 1,088	600	0.80	56.50	42.70
	Average		1.97	52.20	45.83

Table 2 Video sequences resolution and frames types info

Name	Resol.	fps	I	P	B	Total
Bus	352 × 288	25	1	75	74	150
Foreman	352 × 288	25	2	161	137	300
Waterfall	352 × 288	25	2	116	142	260
Johnny	854 × 480	60	3	151	446	600
Basketball	854 × 480	50	2	250	248	500
Cactus	854 × 480	50	2	249	249	500
Johnny	1,280 × 720	60	3	151	446	600
Basketball	1,280 × 720	50	2	247	251	500
Cactus	1,280 × 720	50	2	244	254	500
Basketball	1,920 × 1,088	50	2	236	262	500
Cactus	1,920 × 1,088	50	2	181	317	500
Terrace	1,920 × 1,088	60	3	231	367	600

frames are available typically every 1 s in a video sequences in order to overcome communication errors and their propagation when data are lost during transmission. However, a high number of IDR frames will significantly impact the parallel efficiency and the scalability of our algorithm. The minimum interval between IDR frames is typically equal to the frame rate (as in the default settings of the x264 encoder [14]). For example, an HD video sequence with a frame rate of 60 frames per second (fps) will have an IDR frame every 60 frames (equivalent to 1 s). We can increase or decrease the frequency of IDR frames in the encoder configuration. However, a high frequency of IDR frames, for example one I-frame every 10 frames, decreases the compression efficiency of the encoder, and the visual results will not be noticeable by the human vision. The default configuration for the IDR period in the x264 [14] and the JM [19] H.264 encoders is set to an adaptive decision, which basically inserts an IDR whenever a scene changes. We use this feature in our experiments in order to encode the video benchmarks. The numbers of I-, P-, and B-frames are listed in Table 2 on page 10. The IDR period for low frame rates (25 fps) is around 150 (6 s) and for high frame rates (50–60 fps) is 200–250 (3–5 s).

4.4 Macroblock dependency check algorithm

The macroblock dependency check algorithm is straightforward and simple to implement. Given a list containing all the macroblocks in a slice, a loop that iterates over all macroblocks flags all I-MBs and assigns each remaining macroblock to a group specific for an available core. Then, the groups of macroblocks are decoded in parallel. When all macroblock groups are processed, a loop iterates over all I-MBs that were flagged initially. All the macroblocks in the I-MBs list are decoded sequentially. I-MBs can be processed in parallel if they are not neighbors, meaning they do not have any dependencies between them. The number of I-MBs is not significant in P-frames and B-frames as shown in Table 1 on page 7. If we assign one macroblock to a different

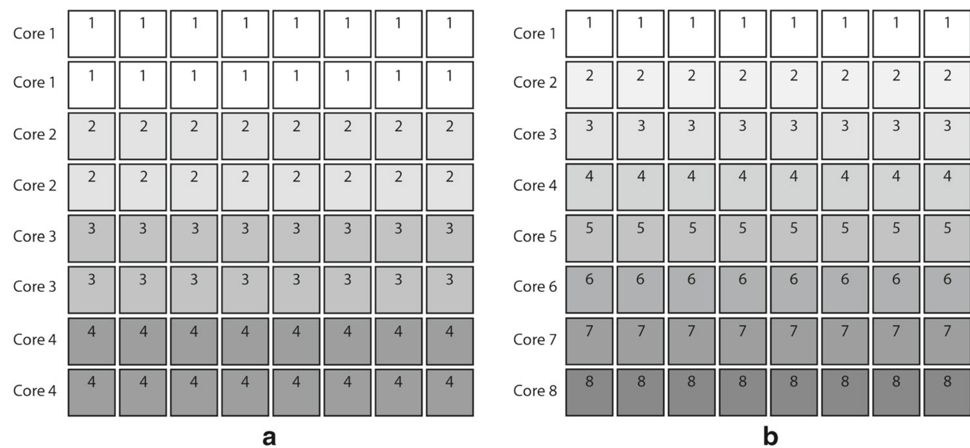

Fig. 5 Parallel decoding of macroblocks mapped to **a** 4 cores and **b** 8 cores

core, the workload is not very important and synchronization overhead will also be added. So we just execute them sequentially in our experiments for the reasons of simplicity and less communication overhead. With the previously mentioned steps, inter-prediction and intra-prediction stages are completed. The output of this stage complies fully with the H.264 standard [8], which means that the output is exactly the same when sequential execution is performed. Decoded macroblocks are then submitted to the deblocking filter in order to make these edges between macroblocks smooth and nearly invisible.

4.5 Macroblocks partitioning

In the parallel decoding algorithm described above, groups of macroblocks are decoded in parallel. In this part, we explain why we chose groups of macroblocks to be decoded in parallel. As explained above, while iterating over macroblocks in a frame slice, we skip intra-prediction macroblocks (I-MBs) and we decode inter-prediction macroblocks (P-MBs and SKIP-MBs) in parallel on multiple cores. Depending on the number of available cores, we group rows of macroblocks in order to be decoded in parallel. The slice is divided by the number of cores horizontally.

Seitner et al. [17] compare 6 parallel representations in terms of stall time and core usage. Among the presented data partitioning approaches, our partition is similar to the slice-parallel splitting approach that is described in [17]. As shown by the authors, this approach has significant stall time overhead, which is caused by synchronization procedures in order to satisfy macroblock dependencies. However, with our approach for satisfying dependencies between macroblocks, the stall time overhead does not apply. We chose this method because of data locality and also due to minimal data transfer initiation overhead. For example, in order to execute a slice of 80 rows of macroblocks on 4 cores processor, each core decode a chunk of 20 rows of macroblocks. Using this partition method, data are only transferred 4 times to the local cores caches. This number of transfers is minimal because it is equal to the number of available cores. Communication overhead between caches of different cores is required when I-MBs depend on other macroblocks that are processed by another core. In Fig. 5, we show an example of a frame of size 8×8 MB (s64×64 pixels) mapped on 4 cores in 5-a and on 8 cores in 5-b. The numbers inside the squares are the numbers of cores. Macroblocks in Fig. 5 are assumed to be all P-MBs or B-MBs. I-MBs are not displayed for illustration purposes.

In a sequential implementation, macroblocks are processed in raster scan mode, starting from top to bottom rows and for each row from left to right macroblock. All independent macroblocks in a slice can be processed at the same time. However, the level of parallelism is limited by the number of available cores. In our parallel implementation, we choose to group macroblocks in rows because it offers a good load balance on different cores. In addition, this level of parallelism has a low synchronization overhead between cores, and it can be considered simple to implement and to manage. Moreover, decoding independent macrobocks vertically or diagonally did not show any significant difference with horizontal decoding because all these macroblocks depend on previously decoded macroblocks. Further studies will be performed in order to group macroblocks based on their dependencies to previously decoded macroblocks. In this paper, we limit our study to the row-based algorithm that is tested on a embedded multicore processor.

4.6 Scalability of parallel motion compensation

In our approach, the highest scalability level is the maximum number of independents macroblocks in a frame slice. Once the dependency detection algorithm isolates the I-MBs, all remaining macroblocks can be processed at the same time. However, the level of parallelism is limited by the available cores in a multiprocessor chip. The optimal speedup will always be when all the macroblocks are assigned to the avail-

able parallel cores. This will eliminate the context switching overhead, which affects the performance in general. For many-core processors, an important limitation that remains unsolved is the huge data communication overhead between cores. For vector processors or general-purpose graphical processing units (GPGPUs) which offer a very high level of parallelism, great potentials exist that may also benefit from the high scalability of our approach. In this paper, we limit our experiments and results to embedded multicore processors.

4.7 Parallel deblocking filter

The deblocking filter, last stage of the H.264 decoder, makes the edges between macroblocks smoother, and thus, it decreases the artifacts that appear when a slice is partitioned into macroblocks. This final stage of the decoder that consists of 41–45 % of the total decoding time as illustrated in Fig. 2 on page 5 is also modified to execute in parallel on different cores. However, dependencies between macroblocks in this stage are different than the dependencies of motion compensation and intra-prediction. During the deblocking filter stage, each macroblock requires that the top and the left macroblocks are already filtered. Figure 6 illustrates the sequential (a) and the parallel (b) filtering modes that are applied on macroblocks in a slice. Both scanning modes satisfy the dependencies requirements of the deblocking filter stage. In Fig. 6a, one marcoblock is filtered at a time. In Fig. 6b, macroblocks colored in dark gray are processed on different cores in parallel. This method, also known as wavefront scheduling, is considered as a commonly used approach for processing independent macroblocks. It can be applied at the intra-prediction, the motion compensation, and the deblocking filter stages as proposed and explained by Zhao et al. [27].

We implement the wavefront parallel method for the deblocking filter stage only. This method satisfies the dependencies requirements of the deblocking filter process as illustrated in Fig. 6b. We implement this parallel processing approach in order to complement our proposed parallel motion compensation algorithm. Both stages process independent macroblocks in parallel. In the following section, experimental results will be provided for the complete parallel implementation of the motion compensation and the deblocking filter stages.

5 Experimental results

In this section, we evaluate our H.264 parallel implementation on a multicore embedded processor. We describe the configuration environment for the real-time execution and the tools that were used to collect all execution information. We gather real-board execution time and energy consumption statistics. We also compare our results with similar literature for parallel H.264 implementations.

5.1 Parallel execution

Parallel execution is considered as a major potential solution for complex applications where sequential execution bounds the performance of these applications. Most processors that are currently available in the market have multiple cores. Applications with high computational complexity may benefit from potential speedup from multiple cores when data or functional parallelism is applicable. Even optimized implementations can still take advantage from parallel techniques. In our research, we choose the H.264/AVC video decoder as our multimedia application benchmark for which we provide a parallel implementation using our innovative approach. We further gather execution statistics and compare results to other relatively similar implementations. In our H.264 parallel implementation, the motion compensation (MC) stage for each row of inter-prediction macroblocks (P-MB) is executed in parallel on different cores. We experiment our parallel implementation using video sequences with CIF (352 × 288), WVGA (854 × 280), HD (1,280 × 720), and FHD (1,920 × 1,080) resolutions on

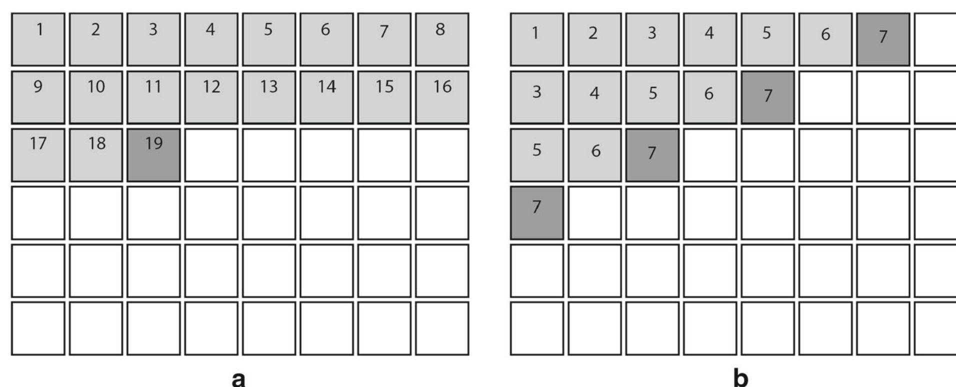

Fig. 6 Sequential and parallel deblocking filter of macroblocks in the H.264 decoder. **a** Sequential. **b** Parallel

an embedded multicore processor. Macroblock dependencies in the same picture slice are avoided by decoding intra-prediction macroblocks (I-MBs) when all other macroblocks of the same slice are already decoded. Overheads emerged as a result of shared memory communications and synchronization between cores. We collect execution time and energy consumption statistics using experiments on a real board with an embedded multicore processor. A virtual threshold for the speedup to the number of cores ratio is identified when large numbers of threads are used.

5.2 Environment and configurations

Our H.264 parallel implementation described in Sect. 4 is executed and tested on a Cuda Development Kit platform [13] with an ARM Cortex-A9 processor with 4 cores [2]. The processor has a memory size of 2 GB and an L2 cache size of 1MB. L1 instruction and data caches both have the size of 32 KB. The maximum frequency is 1.3 GHz when 4 cores are used. This high-end and low-power processor is currently available in many portable devices such as smartphones, tablets, and notebooks. We execute our parallel H.264 decoder using 2, 4, 6, 8, 12, and 16 threads. Each thread is mapped automatically by the operating system (Ubuntu in our case) to a different core. When the number of threads is more than 4, context switching is required to run all threads that are created by the application. We gather statistics using 4 different resolutions: CIF, WGVA, HD, and FHD. With each resolution, we use 3 different video sequences with different image complexities in terms of movement speed and number of objects. Table 2 lists all the video benchmarks that were used in our experiments. The information in Table 2 includes the resolution, the rate of frames per second, the number of I-frames, the number of P-frames, the number of B-frames, and the total number of frames. Real-time execution for all the above video sequences is performed. Execution time is simply calculated by the application and the operating system. Energy statistics are collected by a power measuring instrument, the Agilent LXI digitizer [21]. The digitizer accurately measures the static and dynamic current consumption across the resistors place. The Agilent Technologies L4532A [21] is a high-resolution, standalone LXI digitizer. It offers 2 channels of simultaneous sampling at up to 20 mega samples per second (MSa/s), with 16 bits of resolution. Inputs are isolated and can measure up to 250 volts to handle the most demanding applications. Time and energy results are illustrated and analyzed in the following subsections.

5.3 Results for parallel motion compensation

Experiments are preformed on the videos sequences listed in Table 2. The number of parallel rows of macroblocks increases with the video resolution. Thus, high resolutions scale better than low resolutions with the number of core due to higher number of macroblocks in each frame. Experiments are conducted using 2, 4, 6, 8, 12, and 16 threads on an ARM Cortex-A9 with 4 cores [2]. Figure 7 shows the average speedup of the motion compensation stage for every resolution for different number of threads. For the CIF resolution, the maximum speedup of 1.7 is attained using 4 threads. The speedup decreases as the number of threads increases due to large data communication overhead. HD and FHD video sequences have a speedup higher than 3 with 4 threads where each thread is mapped to different core. The best speedup to the number of threads ratio is when 4 threads are used. The ratio of speedup to number of threads for high-definition resolutions is around 0.8 when 4 threads are used. Doubling the number of threads drops the ratio to 0.6 which cannot be considered as efficient as expected when running a parallel application on a multicore processor. Using a number of threads that are more than the number of cores causes the scheduler to assign more than one thread for one core. Thus, the resulted context switching does not increase the efficiency of the application as shown in our results.

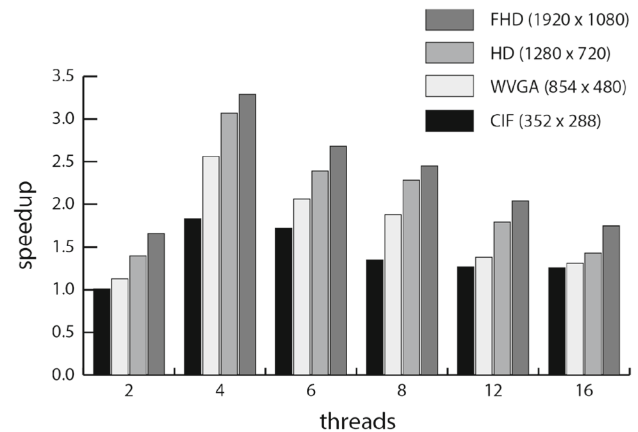

Fig. 7 Speedup of H.264 parallel execution of the motion compensation stage

Results for high resolutions in general have better speedups. This is mainly due to greater workload for each core than smaller resolutions. A larger workload reduces the impact of synchronization and data transfer between cores. One of the reasons is less dependencies between macroblocks being processed on different cores. Another reason is the data transfer overhead, which is required for sending data to different cores. Synchronization also adds an overhead, which is independent of the video resolution. Thus, speedup will be much more efficient for higher resolutions.

5.4 Comparison with related work

For the 2D-wave approach described in [11], the speedup using 4 cores is 2.6 and the highest speedup is around 9.5

Table 3 Comparison of macroblock parallelism scalability with dynamic 3D-wave in [10]

Res.	Tot. MBs	3D-MBs	Par-MBs	Diff. (%)
SD	1,620	1,288	1,592	+23.6
HD	3,600	2,886	3,528	+22.3
FHD	8,160	5,819	7,917	+36.1

using 24 cores. Our results have a better ratio between the speedup and the number of cores; however, we can only compare the speedup up to 4 cores. In addition, our approach has a higher theoretical speedup as the number of independent macroblocks that can be processed at the same time is higher. When processing macroblocks simultaneously, workload on different cores is almost equal. On the other side, when applying the wavefront approach in [11] and [27], the number of independent macroblocks reaches its maximum only when almost half of all macroblocks of the current slice are already decoded. Furthermore, the experimental environment is not the same. We are testing our parallel implementation on a real platform; on the other side, most results in other researches, like [17] and [11], use simulators. In following sections, we will show simulated results for the overall execution of the parallel H.264 decoder.

Exact comparisons with related work cannot be accurate for several reasons like decoder implementation, processor configurations, and video resolutions. However, a comparison of the macroblock scalability between our approach and the dynamic 3D-wave [10] is shown in Table 3. The 3D-wave paper [10] performed a detailed analysis of the parallel scalability of macroblocks. We intend to compare the maximum number of macroblocks that can be processed in parallel between our approach and the dynamic 3D-wave approach. Three video resolutions are being compared. SD resolution (720×576) is compared to WVGA (854×480) because it has the same total number of macroblocks per frame. The remaining resolutions being compared are HD and FHD. The second column lists the total number of macroblocks per frame for each video resolution. The third column displays the average of the maximum number of parallel macroblocks of the four video benchmarks listed in Table 4 in [10]. The fourth column shows the total number of macroblocks per frame that can be processed in parallel using our parallel motion compensation algorithm. Finally, the last column is the difference of the level of parallel macroblock scalability between both approaches. A difference of 22 till 36 % is calculated in favor of our approach. In addition, all parallel macroblocks using our approach are in the same frame. Whereas, in the 3D-wave approach [10], parallel macroblocks are from several frames that are being processed concurrently. We note that the numbers in Table 3 are maximum values which, in practice, cannot be effectively executed in parallel using today's many-

Table 4 Speedup of video sequences executed with multiple threads on multicore processors

Seq/Threads	2	4	6	8	12	16
CIF-Bus	0.94	1.40	1.34	1.23	1.12	1.09
CIF-Foreman	0.85	1.30	1.35	1.10	1.13	1.01
CIF-Waterfall	0.82	1.58	1.40	1.27	1.12	1.08
WVGA-John.	1.06	2.15	1.74	1.65	1.26	1.05
WVGA-Bask.	1.13	1.92	1.68	1.62	1.35	1.14
WVGA-Cact.	1.07	1.81	1.62	1.56	1.29	1.10
HD-Johnny	1.27	2.42	1.91	1.93	1.60	1.36
HD-Basket	1.28	2.14	1.81	1.81	1.58	1.39
HD-Cactus	1.26	2.09	1.78	1.78	1.55	1.34
FHD-Basket	1.40	2.26	1.93	1.89	1.68	1.52
FHD-Cactus	1.42	2.28	1.93	1.86	1.68	1.51
FHD-Terrace	1.44	2.33	1.97	1.93	1.72	1.53

core systems. We choose the group parallel macroblocks in groups of rows depending on the number of available cores in a multicore architecture.

5.5 Results for parallel deblocking filter

Similarly, to the motion compensation experiments, we gather statistics results of our parallel implementation of the deblocking filter using the wavefront algorithm. For the deblocking filter, the wavefront algorithm is the best known parallel algorithm that satisfies the dependency constraints of this stage. The same videos sequences that are listed in Table 2 are used. As described previously, the wavefront algorithm reaches the highest number of independent macroblocks that can be filtered in parallel when the diagonal divides the slice into almost two equal partitions. Parallel deblocking achieves a speedup of 1.44 using 4 threads for CIF resolution and a speedup of 2.6 using 4 threads for Full-HD resolution. Figure 8 displays the average speedup results

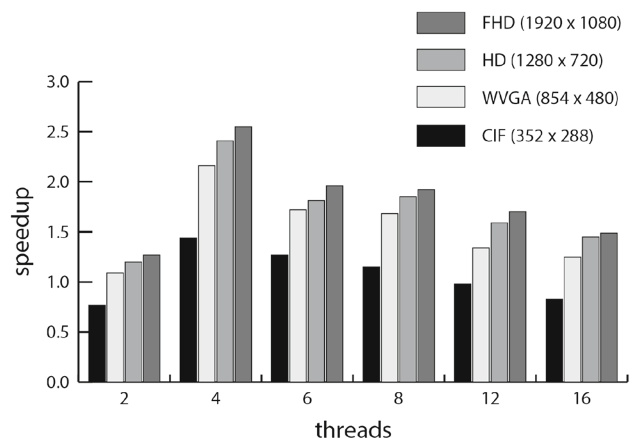

Fig. 8 Speedup of H.264 parallel execution of the deblocking filter

for different resolutions and different number of threads. As mentioned earlier, the scalability of the wavefront algorithm is not as high as our parallel decoding algorithm for motion compensation and intra-prediction stages. In addition, the workload for every core using the wavefront algorithm is only one macroblock, whereas the workload of the motion compensation algorithm is composed of many macroblocks depending on the number of available cores. A smaller workload also adds more synchronization overhead. Thus, the speedups of the parallel deblocking filter are lower than the motion compensation speedups displayed in the previous subsection.

5.6 Results for overall H.264 execution

Our main goal is to optimize all the stages the H.264 decoder. We apply parallel techniques for the motion compensation and the deblocking filter stages. On the other hand, the entropy decoder stage is inherently sequential. Thus, parallel techniques are very hard to apply or sometimes impossible due to its specification requirements. We collect execution time and energy consumption statistics for the proposed H.264 parallel implementation. The fractions of the different stages vary among different video sequences. As a result, the overall performance is considered as a weighted average of all speedups based on the average percentage of each phase.

Figure 9 illustrates the overall speedups attained for the complete execution of the decoder with the described optimization techniques. The total speedups of 1.4, 2.0, 2.2, and 2.3 are reached using 4 threads on 4 cores for the resolutions CIF, WVGA, HD, and FHD, respectively. The detailed results for every video sequence are listed in Table 4. The sequential execution of the entropy decoding stage which is about 14–19 % of the overall decoding scales down significantly the overall speedup. This stage may be enhanced by implementing a hardware version of the entropy decoder. FHD resolutions have the highest speedup because of their large frame sizes. All maximum speedups are attained using 4 threads on 4 cores. This is mainly due to the absence of context switching where each thread is mapped to one core. Using more than 4 threads will require the operating system to assign more than one thread to a core causing context switching, and as a result, more overhead and stall time will be added to the overall execution. Only CIF video sequences have speedups less than 2 when 4 threads are mapped onto 4 cores. The ratio of speedup to the number of cores is therefore around 0.6. This leads us to conclude that high resolution benefits more from multicore processors than lower resolutions. So Full-HD resolutions have the best speedup with higher number of cores. 4K resolution appeared recently in high-end TVs and in movies theaters. These high resolutions will further benefit from many-core processors as huge amounts of data will require more processing power.

Energy measurements for the complete execution are displayed in Fig. 10. The best energy saving results correspond to the FHD resolutions using 4 threads, which attain 63 %. These results are also measured for the complete execution of the optimized decoder. For 12 and 16 threads, energy consumption will increase compared to sequential execution.

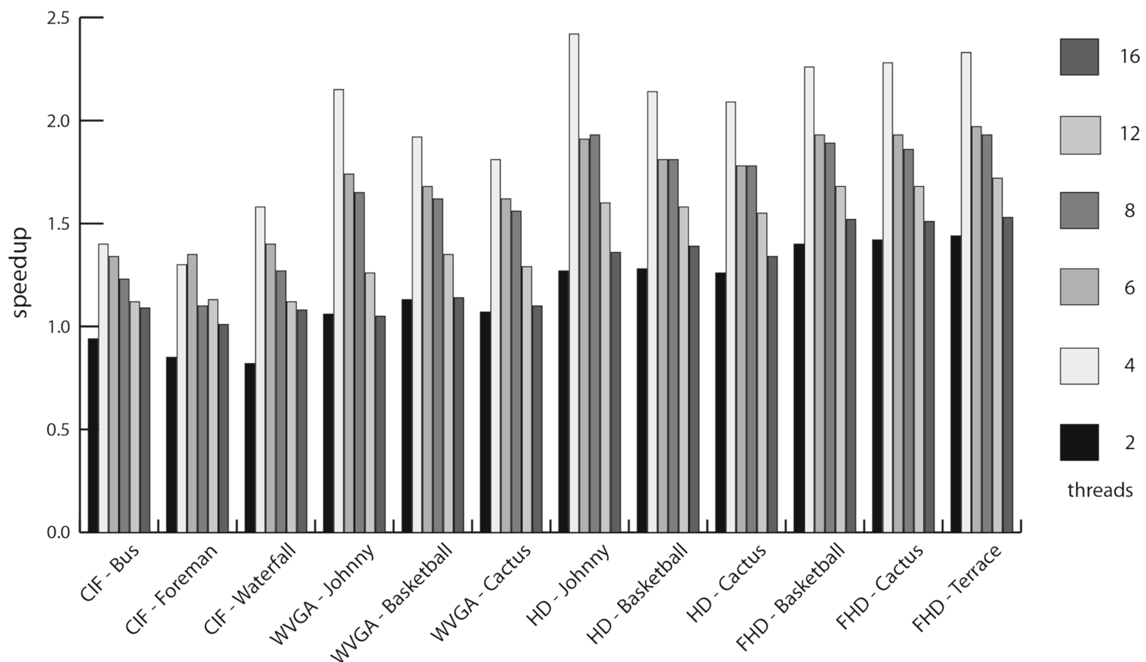

Fig. 9 Total speedup for the complete decoding process on multicore processor

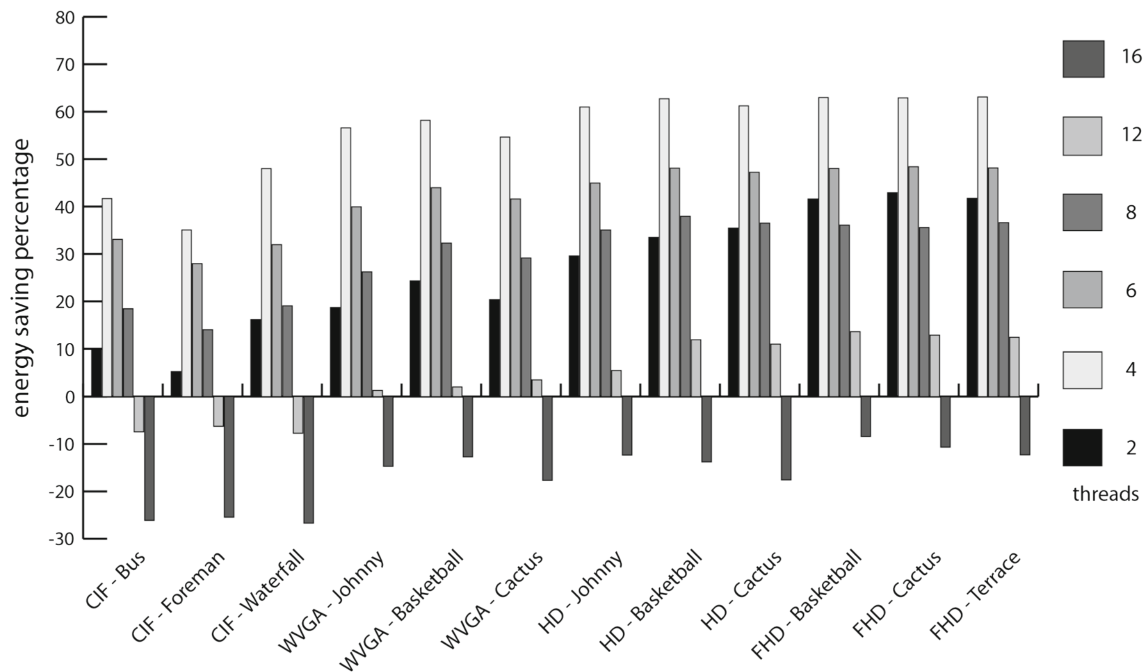

Fig. 10 Total energy saving for the complete decoding process on multicore processor

Thus, we conclude that energy saving does not scale linearly with the number of threads or cores.

5.7 Simulated H.264 execution

As a complementary step to experiment our parallel H.264 algorithm, we execute our implementation on the multicore simulator Multi2Sim [23]. Figure 11 shows the speedup of our parallel H.264 implementation on 2, 4, 8, 16, and 32 cores. HD and FHD resolutions are used with the Baseline and the Main profiles. These results display the average of the three video sequences listed in Table 2. On 2 cores, the speedup for Baseline profile is 1.7 and 1.5 for Main profile. The speedup increases with the number of cores; however, this increase is not linear. Using 32 cores, the speedup reaches 5.2 for the Baseline profile and 3.2 for the Main profile. The difference between both profiles becomes more significant as the number of cores increases. The time needed for motion compensation and deblocking filtering in the Baseline profile is higher than the Main profile. The entropy decoding execution time is less for the Baseline profile compared to the Main profile. This is mainly due to the CABAC algorithm for the entropy decoder, which is used in the Main profile. CABAC has a better compression at the expense more complexity. Thus, our parallel method is better exploited with the Baseline profile where the entropy decoder, which is executed sequentially, has less impact on the overall speedup. The parallel scalability of our H.264 decoder is significantly affected by data communication between cores. The results are shown in Fig. 11 for 8 cores and more are far from the theoretical speedup. If we calculate the ratio of speedup to the number of cores, we can see that the ratio is 0.85 on 2 cores and 0.65 on 4 cores. For higher numbers of cores, the ratio is below 0.5, which is considered very inefficient and unworthy of parallel execution. Many-core processors with 8 or more cores should have special memory architecture than dual and quad cores processors. Thus, parallel algorithms, like our H.264 parallel implementation, should be adapted to benefit from many-core processors and to minimize data communication overhead imposed by a large number of parallel cores.

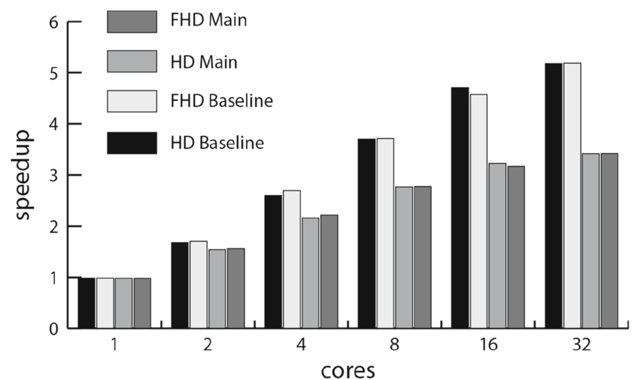

Fig. 11 Speedup of H.264 parallel execution using the Multi2Sim simulator

5.8 Theoretical speedup

Figure 12 shows the theoretical speedup that can be reached for the overall execution of the H.264 parallel decoder. The

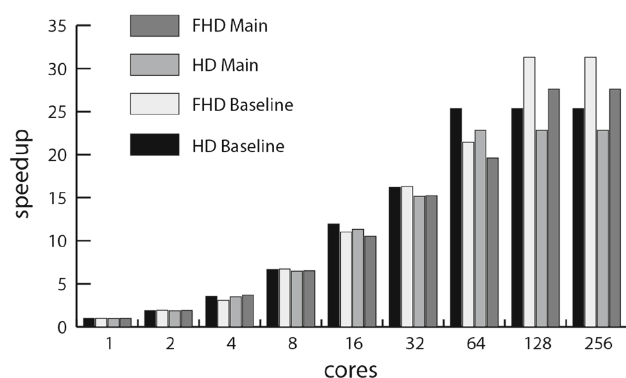

Fig. 12 Theoretical speedup of H.264 parallel execution without data communication overhead

differences with the simulated execution results displayed in Fig. 11 are relatively small up to 8 cores. For 16 cores, the speedup of our parallel H.264 algorithm should be around 12. The speedups keep increasing until 64 cores for HD resolutions and 128 cores for FHD resolutions. This threshold appears when the number of cores will become more than the number of macroblock rows. However, using our algorithm for parallel motion compensation, the granularity can become smaller so that we can benefit from additional cores. If the number of cores is close to the number of parallel macroblocks that are listed in Table 3, then the speedup would become much higher. In real many-core architecture, this speedup comes with a huge memory communication overhead that affects the speedup dramatically. New parallel processing architectures should be used for such high levels of parallelism. This issue is still a major bottleneck in the computing industry. In our research, we also aim to explore and to experiment new parallel architectures in order to show to the full benefits of parallel computing.

6 Conclusion and future works

We have introduced a novel parallel technique for H.264 video decoder parallel optimization. Our approach decodes groups of macroblock rows of the H.264 decoder in parallel with an algorithm that detects dependencies on the fly based on isolating intra-prediction macroblocks (I-MBs). Experiments using low- and high-definition video sequences show that high resolutions have a better performance when executed in parallel. However, speedup and energy savings do not scale with the number of cores. This limit is mainly due to the increase in data transfer between cores. The best speedup with the highest ratio to the number of cores of the motion compensation parallel implementation is 3.3 for FHD resolutions using 4 threads on 4 cores. A parallel macroblock-based implementation of the deblocking filter is also implemented. An overall speedup of 2.3 is attained for the complete H.264 parallel implementation. Our optimized decoder is tested on a real device with an ARM Cortex-A9 processor with 4 cores. Our parallel algorithm is also tested on a mutlicore simulator in order to explore to scalability of our algorithm on multiprocessors up to 32 cores. We plan to test our H.264 parallel implementation on more recent processors with larger number of cores as we will also explore more optimization techniques for different multimedia applications. Future work will also cover experiments on vector processors in order to benefit from the high scalability levels and simplicity that our approach offers.

References

1. Apple: iphone. http://www.apple.com/iphone/ (2013)
2. ARM-ltd.: Cortex-a9 processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php (2012)
3. ARM-ltd.: Cortex-a15 processor. http://www.arm.com/products/processors/cortex-a/cortex-a15.php (2013)
4. Chong, J., Satish, N., Catanzaro, B., Ravindran, K., Keutzer, K.: Efficient parallelization of h.264 decoding with macro block level scheduling. In: IEEE International Conference on Multimedia and Expo 2007, pp. 1874–1877 (2007). doi:10.1109/ICME.2007.4285040
5. FFmpeg: Ffmpeg Project. http://www.ffmpeg.org/ (2012)
6. Gurhanli, A., Chen, C.P., Hung, S.H.: Gop-level parallelization of the h.264 decoder without a start-code scanner. In: 2010 2nd International Conference on Signal Processing Systems (ICSPS), vol. 3, pp. V3-627–V3-630 (2010). doi:10.1109/ICSPS.2010.5555416
7. Horowitz, M., Joch, A., Kossentini, F., Hallapuro, A.: H.264/avc baseline profile decoder complexity analysis. IEEE Trans. Circuits Syst. Video Technol. **13**(7), 704–716 (2003). doi:10.1109/TCSVT.2003.814967
8. ITU-T, ISO/IEC: Advanced Video Coding for Generic Audiovisual Services. ITU-T Rec. H.264 (2012)
9. Kannangara, C.S., Richardson, I.E.G., Bystrom, M., Solera, J., Zhao, Y., Maclennan, A.: Complexity reduction of h.264 using lagrange optimization methods. In: IEE VIE 2005, Glasgow, UK (2005)
10. Meenderinck, C., Azevedo, A., Juurlink, B., Alvarez Mesa, M., Ramirez, A.: Parallel scalability of video decoders. J. Signal Process. Syst. **57**(2), 173–194 (2009). doi:10.1007/s11265-008-0256-9
11. Mesa, M.A., Ramirez, A., Azevedo, A., Meenderinck, C., Juurlink, B., Valero, M.: Scalability of macroblock-level parallelism for h.264 decoding. In: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems (ICPADS '09), pp. 236–243. IEEE Computer Society, Washington, DC, USA (2009). doi:10.1109/ICPADS.2009.124
12. Nishihara, K., Hatabu, A., Moriyoshi, T.: Parallelization of h.264 video decoder for embedded multicore processor. In: 2008 IEEE International Conference on Multimedia and Expo, pp. 329–332 (2008). doi:10.1109/ICME.2008.4607438
13. nVIDIA: The Cuda Development Kit from Seco (2012). http://www.nvidia.com/object/seco-dev-kit.html
14. Organization, V.: x264 Encoder (2013). http://www.videolan.org/developers/x264.html
15. Pieters, B., Hollemeersch, C.F., De Cock, J., Lambert, P., De Neve, W., Van De 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). doi:10.1109/TCSVT.2011.2105553
16. Samsung: Samsung Smartphones (2013). http://www.samsung.com/fr/consumer/mobile-phones/smartphones/
17. Seitner, F.H., Bleyer, M., Gelautz, M., Beuschel, R.M.: Evaluation of data-parallel h.264 decoding approaches for strongly resource-restricted architectures. Multimed. Tools Appl. **53**(2), 431–457 (2011). doi:10.1007/s11042-010-0501-7
18. Sihn, K.H., Baik, H., Kim, J.T., Bae, S., Song, H.J.: Novel approaches to parallel h.264 decoder on symmetric multicore systems. In: Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '09), pp. 2017–2020. IEEE Computer Society, Washington, DC, USA (2009). doi:10.1109/ICASSP.2009.4960009
19. Suehring, K.: H.264 Reference Software (2012). http://bs.hhi.de/suehring/tml/
20. Sullivan, G., Ohm, J., Han, W.J., Wiegand, T., Wiegand, T.: Overview of the high efficiency video coding (hevc) standard. IEEE Trans. Circuits Syst. for Video Technol. **22**(12), 1649–1668 (2012). doi:10.1109/TCSVT.2012.2221191
21. Technologies, A.: High-Resolution lxi Digitizers (2012). http://www.home.agilent.com/en/pd-1445167-pn-L4532A/
22. Tudor, P.N.: Mpeg-2 video compression. Electron. Commun. Eng. J. **7**(6), 257–264 (1995). doi:10.1049/ecej:19950606
23. Ubal, R., Jang, B., Mistry, P., Schaa, D., Kaeli, D.: Multi2sim: a simulation framework for cpu-gpu computing. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12), pp. 335–344. ACM, New York, NY, USA (2012). doi:10.1145/2370816.2370865
24. VanDerTol, E., Jaspers, E., Gelderblom, R.: Mapping of h.264 decoding on a multiprocessor architecture. In: Proceedings of the SPIE Conference on Image and Video Communications and Processing, pp. 707–718 (2003)
25. Wang, S.W., Yang, S.S., Chen, H.M., Yang, C.L., Wu, J.L.: A multicore architecture based parallel framework for h.264/avc deblocking filters. J. Signal Process. Syst. **57**(2), 195–211 (2009). doi:10.1007/s11265-008-0321-4
26. YouTube: Youtube Advanced Encoding Settings (2013). https://support.google.com/youtube/answer/1722171
27. Zhao, Z., Liang, P.: Data partition for wavefront parallelization of h.264 video encoder. In: Proceedings. 2006 IEEE International Symposium on Circuits and Systems, 2006 (ISCAS 2006), p. 4 (2006). http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1693173