CHAPTER 28

PARALLEL PROGRAMMING FRAMEWORK FOR H.264/AVC VIDEO ENCODING IN MULTI-CORE SYSTEMS

Nuno Roma, António Rodrigues and Leonel Sousa

TU Lisbon / IST / INESC-ID, Lisbon, Portugal

28.1 INTRODUCTION

Among the several multimedia applications that have emerged along the past decade, video encoding has gained a particular relevance in a vast set of domains. However, it is also one of the most computational demanding. In particular, the recognized success of the latest generation of video standards, such as the H.264/MPEG-4 Part 10 (or AVC), is mainly due to its remarkable encoding performance in what concerns the relation between the output video quality and resulting bit-rate, at the cost of a significant increase of the computational complexity. As a consequence, real-time encoding by exploiting the whole set of offered encoding mechanisms is still far beyond the capabilities of most computational systems.

Programming Multi-core and Many-core Computing Systems. By Sabri Pllana and Fatos Xhafa

To cope with such difficulties, several approaches have been proposed that try to take advantage of current parallel platforms to accelerate the encoding [6, 10, 5, 18]. Nevertheless, most of such proposals represent specific optimizations to the considered platforms, requiring the rewrite of the encoder software (SW) whenever a new target hardware (HW) platform or parallelization model is considered.

To circumvent such limitations, a new parallel programming framework is presented. This framework allows to easily and efficiently implement high performance H.264/AVC video encoders in a wide set of different parallel platforms. The offered modularity and flexibility make this framework particularly suited for efficient implementations either in homogeneous or heterogeneous parallel platforms, providing a suitable set of fine-tuning configurations and parameterizations that allow a fast prototyping and implementation, thus significantly reducing the developing time of the whole video encoding system.

28.1.1 H.264/AVC video standard

The H.264/AVC standard has been widely adopted by most recent video applications to address the consumers' needs and the most demanding encoding requirements. The standard is divided in several profiles to define the applied encoding techniques, targeting specific classes of applications. For each profile, several levels are also defined, specifying upper bounds for the bit stream or lower bounds for the decoder capabilities, such as processing rate, capacity of multi-picture buffers, video rate, motion vector range, etc. [21].

To achieve the offered encoding performance, this standard incorporates a set of new and powerful techniques (see Fig. 28.1), namely: 4×4 integer transforms, variable block-size inter-frame prediction, quarter-pixel motion estimation (ME), in-loop deblocking filter, improved entropy coding based on context-adaptive variable-length coding (CAVLC) or on content-adaptive binary arithmetic coding (CABAC) and new intra-frame prediction modes. Moreover, the adoption of bi-predictive frames (B-frames), along with the previous features, provides a considerable bit-rate reduction with negligible quality losses. As a result, when compared with other previous standards (such as H.263, MPEG-1/2 Video or MPEG-4 Visual), the H.264/AVC has proved to provide greater coding efficiency levels, with an excellent tradeoff between the output video quality and bit-rate.

However, the simultaneous exploitation of those new features significantly increased the encoder computational cost. As an example, a direct execution of a straight compilation of the JM reference software [14] in a latest generation processor (running at 2.7 GHz), leads to frame-rate performance levels as low as a single or at most a couple of 4CIF frames per second. At this respect, several complexity analysis have shown that the *Inter prediction* module is usually the most time consuming - about 80% - followed by the *Interpolation* module [1].

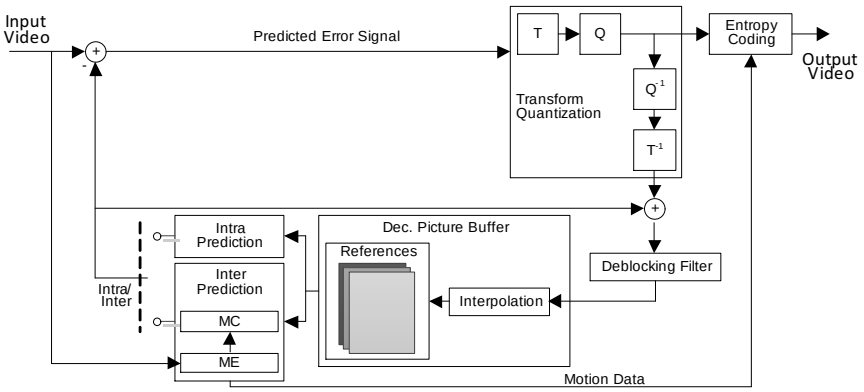


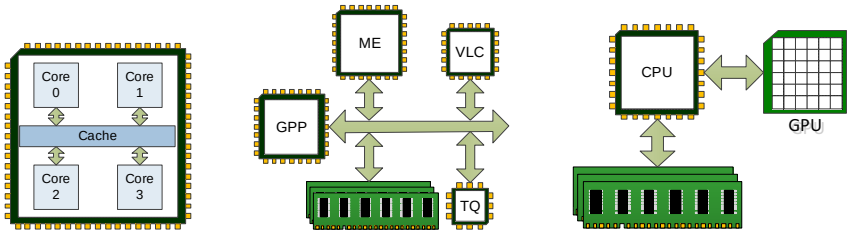
Figure 28.1 H.264/AVC encoding loop.

28.1.2 Parallel architectures and platforms for video coding

To account for the complexity problem of the H.264/AVC video standard, several different approaches have been adopted, either from the SW point of view (e.g. application of low complexity ME algorithms [13]), or from the HW point of view. In particular, with the vast set of parallel processing platforms that are now available, further levels of parallelism are now worth exploiting, either on *homogeneous* parallel platforms composed by multi-core processing systems with several identical CPUs sharing the same chip (see Fig. 28.2(a)), or on *heterogeneous* platforms [12]. These last alternatives are often implemented either with dedicated processing structures integrated in an embedded system on chip (SoC) [8] (see Fig. 28.2(b)) or even with accelerators composed by graphics processing units (GPUs) interconnected to off-the-shelf general purpose processors (GPPs) (see Fig. 28.2(c)).

In the particular domain of video coding, *homogeneous* parallel platforms are usually applied in the exploitation of data-level parallelism techniques, by distributing the video data to be encoded/decoded across several similar parallel computing nodes. Moreover, with the advent of single-instruction multiple-data (SIMD) vector extensions to the ISA of current processors, these techniques have been even complemented with the exploitation of a sub-word parallelism level, by simultaneously processing several data elements with a single instruction. In contrast, *heterogeneous* platforms often adopt functional/task parallelism techniques, where the several modules of the video encoder/decoder are independently implemented by the different parallel computing nodes. In particular, many of such architectures adopt a pipeline processing scheme, where the video data is sequentially processed by the several different and independent stages of the pipeline.

Until very recently, most parallelization efforts around the H.264 standard have been mainly focused on the *decoder* implementation [20, 5, 2, 3], where



(a) Homogeneous. (b) Heterogeneous, with embedded accelerators. (c) Heterogeneous, with GPU accelerators.

Figure 28.2 Parallel video coding architectures.

the complex data dependencies that characterizes the encoding loop are not observed. When the most challenging and rewarding goal of parallelizing the *encoder* is concerned, it has been observed that a significant part of the efforts were devised in the design of specialized and dedicated systems [16, 9, 15]. Most of these approaches are based on parallel or pipeline SoC topologies, using dedicated HW structures to implement some of the most demanding parts of the encoder, and leaving the remaining sequential and less complex code to be executed in a GPP. In most of such implementations, the corresponding segments of the original SW code are simply replaced by instantiations of the accelerated procedures in the proper HW structures. Other similar approximations make use of heterogeneous structures composed by Digital Signal Processors (DSPs) or very long instruction word (VLIW) processors to accelerate the implementation of the encoding procedure. Some examples of such approach are the TriMedia processors from NXP Semiconductors (former Philips Semiconductors^{TM}) [20] and the OMAP processors from Texas Instruments^{TM} [3]. Nevertheless, and independently of the adopted accelerating structure, difficult challenges still often arise in what concerns the transfer of the processed data, as well as the concurrent access to the shared frame memory by the GPP and the several accelerators, usually requiring complex and platform-specific implementation issues and optimizations.

On the other hand, when *pure-SW* approaches are considered, fewer parallel solutions have been proposed. Most of them are based on the exploitation of data-level parallelism, in order to simplify the schedule and the synchronization of the several processing cores. One popular parallelization approach is based on the massive use of similar and concurrent threads, by exploiting the several CPUs that are currently available in multi-core chips [5]. As it will be seen in section 28.2.1, frames can be divided in several independent slices and an individual thread is assigned to each slice. Some of such strategies even make use of Intel^{TM}hyper-threading (HT) technology to increase the number of concurrent threads [6, 10]. Furthermore, some proposals have even complemented the exploited parallelization by also using SIMD multimedia vector instructions currently available in MMX and SSE extensions [5].

Other parallelization approaches based on a heavy exploitation of similar and concurrent threads make use of the OpenMP pragmas for their implementation [19]. They mostly combine the use of thread queues to process the several segments of pixels, together with the exploitation of HT to further speedup the encoding.

Another approach is based on the use of message passing communication protocols (e.g.: MPI), namely on clusters composed by several independent computers [18]. One common strategy is to implement the encoder architecture in parallel, where an independent group-of-pictures (GOP) can be assigned to each cluster node. Furthermore, each node can even be implemented by a multi-core CPU, allowing further parallelization. However, these solutions often present, as their main disadvantage, significant communication overheads that can even surpass the computation time. Moreover, they also require greater amounts of memory to accommodate the several encoded sub-streams at the same time.

Meanwhile, other parallelization approaches have also arisen by exploiting some recent heterogeneous architectures that emerged in the market [11, 17]. One of such proposals includes the implementation of a pipeline encoding structure in the Cell Broadband Engine [11]. In such implementation, the SPEs are used to exploit both slice-level and macroblock-level (see section 28.2.1) parallelism, achieving real-time processing for high-definition image formats.

Other acceleration approaches have also emerged by using the capability of current GPUs to speedup certain parts of the encoder with data-level parallelism [7]. As an example, in [4] it is presented the implementation of the ME module by using the GPU support and providing a speedup of about 12.

Independently of the adopted strategy, the innumerous data dependencies imposed by this complex video standard frequently inflict a very difficult challenge to efficiently take advantage of the several possible parallelization strategies that may be applied. Moreover, the use of the vast set of powerful parallel platforms that are now available has been often refrained by the absence of an unified parallel encoding framework that easily adapts to and efficiently exploits the set of variable resources offered by such concurrent platforms. In this scope, a flexible and highly modular parallel programming framework for *pure-SW* or *HW-accelerated* H.264/AVC encoders is now presented. The aimed challenge is to speedup the encoding procedure without sacrificing the output video quality or increasing the resulting bit-rate. The conducted evaluations, by using different instantiations of the framework, have shown that linear and close to optimal speedup values, in what concerns the achieved frame-rate, can be obtained in current homogeneous parallel platforms. Moreover, the provided modularity and flexibility attested its configurable attributes, in order to easily and better adapt it to the targeted parallel platform.

28.2 PARALLEL PROGRAMMING FRAMEWORK FOR H.264/AVC VIDEO ENCODING

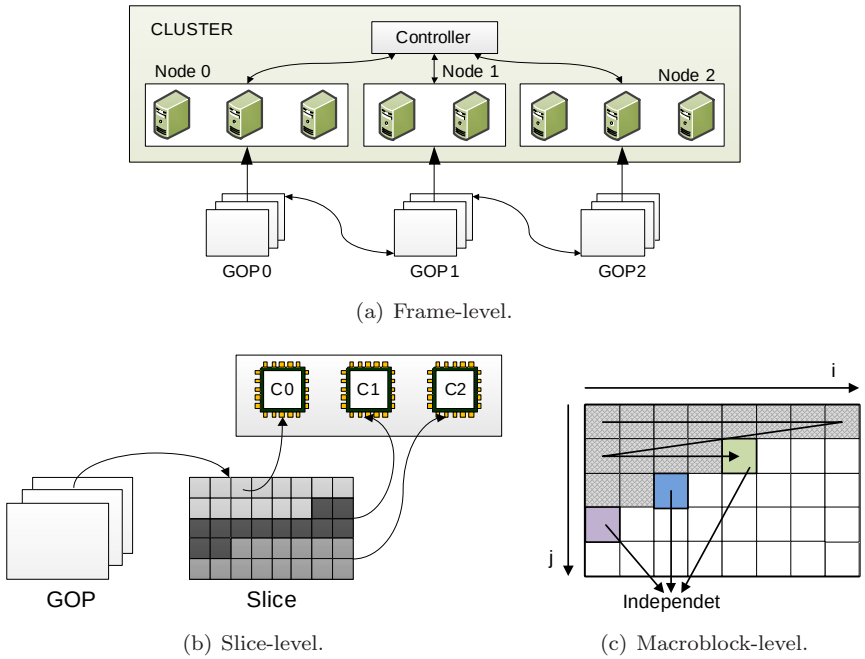
To circumvent the recognized need for a generic and highly modular SW architecture that can be used to efficiently implement H.264/AVC encoders in a vast set of different parallel structures, an innovative parallel programming framework is presented. With such framework, it is given the programmer or system integrator the capability to easily configure and adapt the SW architecture to several different platforms, ranging from the *homogeneous* solutions composed by several GPPs that extensively exploit data-level parallelism, to distinct *heterogeneous* solutions where functional-level concurrency can be exploited in different pipeline/data-flow topologies (see Fig. 28.2).

28.2.1 Data-level parallelism

Several parallelization models have been considered to improve the performance of H.264/AVC encoders [5, 10, 18]. Due to the encoder's nature, many of these parallelization approaches exploit concurrent execution at: *frame-level*, *slice-level*, *macroblock-level*. However, careful design methodologies in what concerns its parameterization and modularity have to be considered, in order to avoid the introduction of performance losses in terms of the final bit-rate and peak signal to noise ratio (PSNR).

At *frame-level*, the input video stream is usually divided in GOPs. Since GOPs are usually made independent from each other, it is possible to develop a parallel architecture where a controller is in charge of distributing the GOPs among the available cores (see Fig. 28.3(a)). The advantages of this model are clear: PSNR and bit-rate do not change and it is easy to implement, since GOPs' independency is assured with minimal changes in the SW code. However, the memory requisites significantly increase, since each encoder must have its own decoded picture buffer (DPB), where all GOPs' references are stored. Moreover, real-time encoding is hard to implement using this approach, making it more suitable, for example, for video storage purposes. As a consequence, this solution has been mainly used in cluster systems [18]. Other parallelism levels usually have to be exploited in order to further improve the speedup.

In *slice-level* parallelism, frames are divided in several independent slices, making the processing of macroblocks (MBs) from different slices completely independent (see Fig. 28.3(b)). In the H.264 standard, a maximum of sixteen slices are allowed in each frame. This approach allows to exploit parallelism at a finer granularity, which is suitable, for example, for multi-core computers where parallel encoding of the several defined slices may be concurrently executed in multiple threads for each individual frame [10]. Moreover, the resulting increase of the allocated memory is smaller, because only one DPB is required. The main issues of this alternative are concerned with the limited number of slices per frame (sixteen in the H.264 standard), together with



(a) Frame-level.

(b) Slice-level.

(c) Macroblock-level.

Figure 28.3 Exploited data-level parallelism models.

a greater parallelization effort in order to ensure a good performance and the need to redesign some data structures and algorithms in order to avoid caching of unnecessary data. Furthermore, this model often restricts the exploited spatial prediction within a frame, thus leading to a moderate increase of the resulting bit-rate.

The parallelism at *macroblock-level* allows independent MBs to be encoded at the same time [2]. According to the standard, a given MB is predicted using its left and upper three neighbors, which can be performed by following a wave-front approach, as depicted in Fig. 28.3(c). Any two MBs are said to be *independent* if there isn't any data dependency in their prediction. As it will be seen in section 28.4.3, this strategy may be used as a viable alternative to complement the exploited parallelism level. The main design issues of this approach are concerned with the need of a centralized control, to guarantee that only independent MBs are processed in parallel, and with the non-uniform distribution of the computational weight that may arise among the cores. However, in middle and high resolution video sequences, as well as when a great number of processors is available, this model may be preferable to the slice-level, since the parallelism is only limited to $\lceil N/2 \rceil$, where N denotes the number of MBs in the diagonal of the frame/slice (see Fig. 28.3(c)).

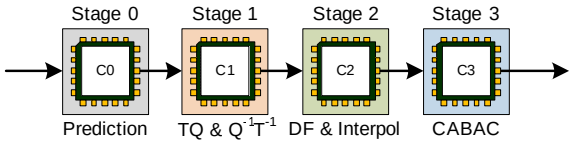
28.2.2 Functional-level parallelism

To ensure the maximum flexibility of the framework, the several modules of the encoder were carefully structured and implemented in independent routines. With such approach, it becomes possible to easily exploit functional-level parallelism, by using pipeline or data-flow topologies where each available core/accelerator may implement a different encoder module. Such feature is particularly important in heterogeneous configurations, where distinct parts of the encoder may be easily migrated to dedicated or specialized architectures.

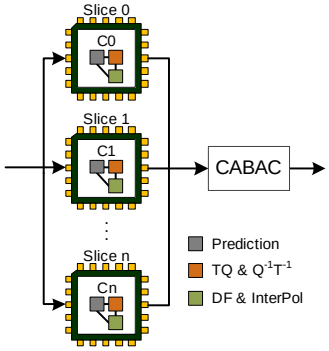
With such SW architecture, three different topologies are made available:

- *Pipeline*
- *Straight parallel*
- *Mixed*

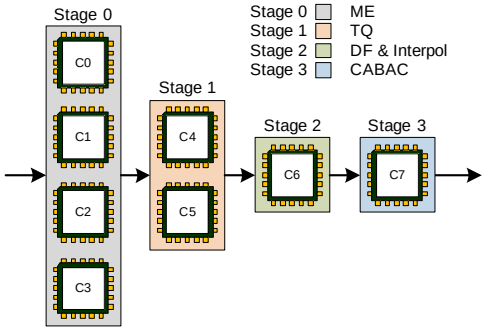
In the *pipeline* topology, illustrated in Fig. 28.4(a), the encoding procedure is divided in several different stages. Each of these stages implements an individual or a particular set of encoding modules. The number of concurrent stages is determined by the number of cores/accelerators available in the system. Nevertheless, for each specific concretization of this topology, three main issues need to be carefully analyzed: how the stages communicate, how they are synchronized and how the several modules are distributed among the stages.



(a) Pipeline.



(b) Straight parallel.



(c) Mixed.

Figure 28.4 Parallel topologies to implement the encoding modules depicted in Fig. 28.1 by using functional-level parallelism.

Table 28.1 Example configuration using a pipelined architecture to implement the eight inter-prediction modes in a variable number of cores.

Number of Cores	Number of Pipeline Stages							
	1	2	3	4	5	6	7	8
2	16×16 16×8 8×16 DIR	8×8 8×4 4×8 4×4						
3	16×16 16×8	8×16, $DIR, 8 \times 8$ 8×4	4×8, 4×4					
4	16×16	16×8 8×16 DIR	8×8, 8×4	4×8, 4×4				
6	16×16	16×8	8×16	DIR, 8×8	8×8, 4×8	4×4		
8	16×16	16×8	8×16	DIR	8×8	8×4	4×8	4×4

Since it is impossible to guarantee that all stages have the same processing time, it is important to ensure that this SW pipeline only advances when all stages have finished their processing. To achieve this synchronization, barrier instances have been adopted in the SW architecture. As soon as all parallel execution flows reach these barriers, they resume executing in parallel the code that follows the barrier. Hence, to better balance the pipeline stages, it is also important to evaluate the processing time corresponding to all sub-functions. Only then should they be grouped in pipeline stages.

Table 28.1 depicts one example configuration. Considering that *inter-frame prediction* represents the most demanding part of the encoder, this example illustrates one possible configuration where the ME corresponding to the eight possible prediction modes were implemented in a homogeneous architecture using a pipeline topology, by considering the usage of 2, 3, 4, 6 and 8 cores.

The design principle corresponding to the *straight parallel* topology is particularly targeted for the exploitation of data-level parallelism, where the slices are assigned to similar concurrent threads that are independently executed. By assuming that slices approximately have the same size, it can be expected that all threads finish their tasks at the same time in homogeneous systems.

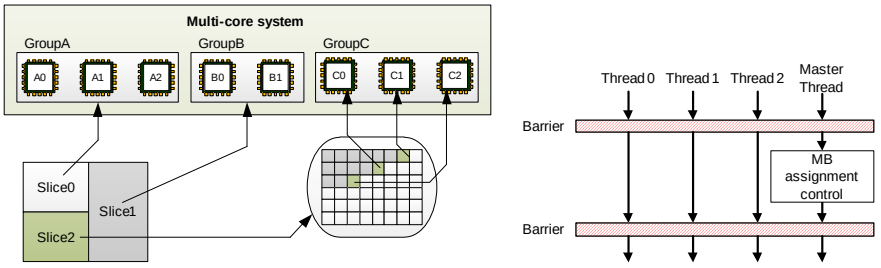
Fig. 28.4(b) illustrates one possible configuration of this topology. In this setup, each available core implements the whole encoding loop, in order to process one particular slice of the video frame. Subsequently, all slice buffers are joined together to supply an external entropy encoding (CABAC/CAVLC) module, in order to form the output stream packet.

The *mixed* topology can be seen as an extension of the pipeline solution, where more than one single core is assigned to implement the most demanding modules of the encoder. Fig. 28.4(c) illustrates one hypothetical configuration of this topology. In this setup, the eight available cores were distributed among

the several modules of the encoder, according to their relative computational requirements: 4 cores to implement the *prediction* module, 2 cores to compute the *transform* and *quantization* (and their corresponding inverses), 1 core to implement the *deblocking filter* and *interpolation* modules and 1 core to implement the *entropy encoding* (CABAC) module.

28.2.3 Scalability

As it was referred before, while pipeline topologies can be particularly adapted to heterogeneous architectures, straight parallel configurations, based on a massive exploitation of *slice-level* parallelism, are often more suited to be implemented in homogeneous systems, composed by several identical CPUs. Nevertheless, according to the H.264/AVC standard such parallelism is limited by a maximum of 16 slices (threads). Hence, to further increase the exploited concurrency, this SW framework allows to complement the applied *slice-level* parallelism with a simultaneous exploitation of *macroblock-level* parallelism. In such configuration (illustrated in Fig. 28.5(a)), a *team of threads* is allocated to the processing of each slice. For each team, independent macroblocks are distributed among the set of cores that were assigned to the processing of that slice. At the end, entropy coding is performed separately and at slice level.



(a) Thread allocation. (b) Thread synchronization.

Figure 28.5 Simultaneously exploitation of slice and macroblock parallelism levels.

The synchronization between the several concurrent threads is guaranteed by using appropriate synchronization barriers. Only threads belonging to the same team are blocked in these barriers. This mechanism is used before and after the execution of the control mechanism that assigns the set of MBs processed by each slice (Fig. 28.5(b)). While the first barrier guarantees that all threads have finished their tasks, the last barrier assures the correct MB assignment. Only after the team master thread has executed the assignment procedure that controls the set of MBs that will be processed in the next run, can the re-running threads of the team start executing. No further synchronization is needed.

Another considered option to increase the exploited concurrency is to adopt *slice scattering*, where slices are divided into several sub-slices located in non-adjacent areas of the frame (see Fig. 28.6). The application of this technique introduces an added level of data independency among the MBs of different sub-slices, allowing them to be processed in different threads. When compared with the previously referred approaches, the extra parallelization level provided by slice-scattering is offered at the cost of an eventual degradation characterized by a slight increase of the resulting bit-rate and a reduction of the output PSNR levels (due to the presence of more blocking effect). As a consequence, this parallel approach is regarded to be more appropriate to encode higher video resolutions, where the spatial redundancy can be exploited without seriously compromising the resulting encoding efficiency.

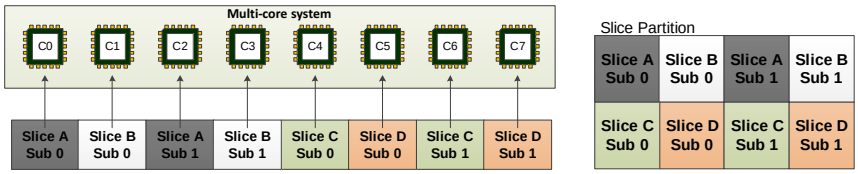


Figure 28.6 Possible slice-scattering distribution in a multi-core architecture.

28.2.4 Software optimization

The presented parallel SW framework is based on JM [14] reference SW, thus maintaining full compliancy with the original encoder. In order to achieve an efficient parallel execution, the conducted research was focused on: *i*) code profiling; *ii*) performance improvement through structures redesign and code optimization; *iii*) definition of the concurrent modules set; *iv*) parallelization.

Code profiling was extensively performed in the first step, in order to identify the most time consuming operations. Several 4CIF standard test video sequences were used for more accurate results (see Fig. 28.7): *Soccer* and *Crew*, characterized by higher amounts of movement; *Harbour* and *City*, with a higher spatial detail. As expected, and according to the results presented in

(a) Soccer. (b) Harbour. (c) Crew. (d) City.

Figure 28.7 Considered set of test video sequences.

Table 28.2 `gprof` profiling results of the H.264/AVC reference SW.

Function	Video Sequences (4CIF)			
	Soccer	Harbour	Crew	City
Inter Prediction	89.1%	86.0%	88.4%	87.7%
Intra Prediction	0.9%	1.1%	0.9%	1.1%
Transf. & Quant.	1.4%	1.7%	1.6%	1.7%
Interpolation	2.3%	2.9%	2.4%	2.7%
Deblocking Filter	0.4%	0.7%	0.5%	0.5%
CABAC	0.4%	0.8%	0.6%	0.3%
Others	6.9%	8.5%	7.2%	7.8%

Table 28.2, *Inter Prediction* is the most computational demanding component of the encoder [1], making it the must suited target for parallelization.

An important step to increase independency and improve the flexibility was the redesign of the original data structures, not only to provide more efficient ways to manipulate and correlate information, but also to save time when fetching them from the memory system, by efficiently exploiting the cache access patterns. On one hand, spatial locality can be further exploited by appropriately resizing the structures, since the probability to store the whole processed data in cache is higher, thus reducing the conflict and capacity misses. On the other hand, temporal locality can also be exploited by data resizing and by joining together the information needed to process wide data sets in each particular module of the encoder. Such resizing was mainly accomplished by removing non-used or duplicated parameters in certain modules and by adjusting their size to their effective range. As it will be seen in section 28.4, when compared with the original data structures, the mutual combination of the conducted optimizations allows a reduction of the required memory space as high as 85.5%.

The conducted code optimization also took into account that many signal processing functions of the encoder can be decomposed into a set of vector operations [5], where the same operation is simultaneously applied to several data elements. By considering that most current processor families and embedded cores already include some multimedia extensions to the instruction set (e.g.: MMX, SSE1, SSE2, SSE3, etc.), such optimization allows to exploit an added degree of SIMD parallelization. In the presented framework, the optional usage of SSE2 SIMD instructions can be activated through a simple compilation option. The usage of these instructions was mainly exploited in the implementation of the most demanding modules, namely, the computation of the sum of absolute differences (SAD) in MB prediction by ME, the Transformation-Quantizer (and corresponding inverses) modules and the Interpolation module (see Fig. 28.1).

To allow the implementation of the presented framework in HW restricted platforms, some additional optimizations and optional configurations were also made available. In this scope, all data structures were statically allocated in memory, allowing this framework to be easily executed in embedded systems that do not necessarily include a dynamic memory allocation system. Furthermore, to ease the implementation in systems with strict memory restrictions, another optimized configuration was also made available, which interchanges the order of the DPB and the Interpolation modules (see Fig. 28.1) for the half-pixel resolutions (keeping the original order for the quarter-pixel resolution frames), thus conferring an extra configurable tradeoff between the required memory resources and the involved computational cost. Such option is particularly suited for pipeline topologies implemented with dedicated accelerators in heterogeneous platforms.

28.3 PROGRAMMING PARALLEL H.264 VIDEO ENCODERS

28.3.1 Framework parameterization

The modularity and flexibility provided by this framework allows an easy customization in order to suit it to distinct types of multi-computer, multi-core or even embedded systems. Such customization can be easily accomplished during source compilation (through the supplied Makefile) by properly choosing the most suitable options to the target system. As an example, the following parameters define the type of data-level parallelism that is exploited:

- CORES - number of considered parallel slices;
- NESTED_CORES - number of considered parallel MBs within each slice.

Likewise, the desired optimization can be selected by the following options:

- SSE_SUPPORT - enables the exploitation of SIMD SSE instructions;
- LOW_MEMORY - enables low memory usage.

Furthermore, thanks to the performed code simplification and division of the encoder into functional modules, the end user can still easily add, remove and modify the sources with minimum effort.

28.3.2 Implementation platforms and APIs

The main aim of the presented framework was to develop a highly modular SW architecture to implement parallel encoders of the latest generation of video standards. To achieve such objective, the several modules of the encoder were implemented with "objects" of self-contained code segments and data structures, in order to provide an easy and efficient migration of such "objects" to several homogeneous or heterogeneous parallel platforms.

A direct consequence of such strict premise is the provided easiness to implement a vast set of different parallel video encoding structures, by using any

of the several parallel Application Programming Interfaces (APIs) currently available, such as MPI, POSIX Threads, OpenMP, OpenCL and CUDA.

Hence, after selecting the data-level and functional-level parallel topology that is most suitable for the considered HW platform, the programmer only has to take care of the migration of the parallelized modules and of the data transfer mechanisms, according to the selected API. Then, proper concretizations of the restricted set of adopted communication structures should be selected. At this respect, several alternatives can be adopted, such as explicit shared memory systems with uniform memory access (UMA) (e.g.: homogeneous multi-core systems implemented either with POSIX Threads or OpenMP), distributed memory systems with non-uniform memory access (NUMA) (e.g: cluster encoding systems implemented with MPI), heterogeneous or non-shared memory systems (e.g.: GPU accelerating systems implemented with CUDA or OpenCL) or even dedicated embedded architectures, implemented with specialized HW structures. Finally, all implicit synchronization mechanisms that are integrated within the framework should be implemented according to the adopted API.

28.4 EVALUATION OF PARALLEL H.264 VIDEO ENCODERS

To demonstrate the feasibility and the advantages provided by the presented framework, several parallel instantiations based on currently available multi-core structures were considered and compared with a sequential implementation of the reference SW running in one core. In particular, considering the specificity and the wide variability presented by most heterogeneous architectures, eventually composed by possibly different accelerating structures, it was decided to adopt a homogeneous structure to demonstrate the performance offered by the proposed framework in an easily reproducible parallel platform. Table 28.3 depicts the characteristics of the considered computational systems. Furthermore, considering that most SW-based and non-dedicated parallel encoders that have been presented up until now make use of homogeneous solutions implemented with POSIX Threads or OpenMP APIs [19, 6, 10, 5], it was decided to evaluate the proposed framework by using a similar environment, in order to achieve fair and correlatable comparisons. As such, *straight-parallel* configurations, exploiting either *slice-level* and *macroblock-level* parallelism will be considered. The whole evaluation procedure was conducted by considering the encoding parameters presented in Table 28.4.

To achieve the most efficient parameterization of the framework, the time consumption profiling results presented in Table 28.2 were carefully considered. From such analysis, it was clear that the *Inter Prediction* module, which includes motion estimation and motion compensation functions, represents the most computational demanding block. As a consequence, the conducted parallelization approach primarily focused on this particular module. Nev-

Table 28.3 Specifications of the considered parallel computational platform.

Platform	Intel^{TM}	AMD^{TM}
Processor	2 × Intel Xeon Quad-Core E5530	8 × AMD Quad-Core 8384
#Cores	8	32
Frequency	2.40 GHz	2.7 GHz
Caches	Individual L1 with 128 KB	Individual L1 with 512 KB
	Individual L2 with 256 KB	Shared L2 with 6 MB
	Shared L3 with 8 MB	–
Memory	24 GB	64 GB
O.S.	64-bits SuSE Linux	64-bits Ubunto Linux
API	OpenMP	OpenMP

Table 28.4 Considered H.264/AVC encoding parameters.

Parameter	**Value**
GOP structure	One I frame followed by thirty B-B-P frames
Intra prediction	All prediction modes
Inter prediction	All prediction modes
Reference frames	3 backward and 1 forward references
ME search algorithm	Simplified UMHexa
ME precision	Quarter-pixel precision
ME error metric	SAD
Entropy coding	CABAC
In-loop deblocking filter	Enabled

ertheless, the whole end-to-end encoder structure was implemented in each instantiation.

28.4.1 Baseline optimizations

As it was referred in section 28.2.4, extensive SW optimizations were considered in order to increase the efficiency of the presented framework. When the memory resources of a particular instantiation of the presented H.264 parallel framework are compared with those of the reference (original) SW, the result of such code improvements becomes clear, leading to a global memory usage reduction of about 93% (see Table 28.5). Such reduction is particularly important when the encoding system is implemented in embedded systems with memory and power consumption restrictions, as well as in parallel configurations that require replication of data in the memory system. Moreover, such optimizations (including the conducted cleaning of the code, reutilization of shared functions and static allocation of data structures) provided a baseline speedup by a factor of 2, even without the exploitation of any level of parallelism.

Table 28.5 Memory allocation for the reference and optimized SW versions.

Data Structure	Reference Software	Optimized Software	
		Regular	Low Memory
Image Parameters	82433 B	248 B	248 B
Input Parameters	5.9 kB	6.1 kB	6.1 kB
Picture Parameters Set	248 B	152 B	152 B
Sequence Parameters Set	2.1 kB	1.7 kB	1.7 kB
Slice	2.3 MB	2.4 MB	2.4 MB
Macroblock	171.7 kB	74.3 kB	74.3 kB
Decoded Picture Buffer	203.1 MB	22.4 MB	6.6 MB
Intra Processing	–	2.0 MB	2.0 MB
Inter Processing	–	2.8 MB	2.8 MB
Total	205.7 MB	29.7 MB	13.9 MB
Memory Saved	–	85.5%	93.2%

Besides such optimizations, some of the most computational intensive modules were wholly re-designed in order to also exploit a SIMD parallelism level, by simultaneously processing several data elements with a single instruction. As an example, the application of MMX vector instructions to the motion estimation (SAD) and the transform modules (DCT) led to partial speedup values of about 1.56 and 1.54, respectively.

28.4.2 Exploiting slice-level parallelism

To evaluate the performance that is offered by the presented framework when *slice-level* parallelism is exploited, each frame of the video sequence under processing was divided into several slices, which were subsequently assigned to an individual core, as described in Fig. 28.4(b).

The results obtained with the Intel platform are illustrated in Fig. 28.8 for the several different parameterizations that are offered by this framework. With the exception of the setup that made use of 16 threads, all the considered cases evidence a speedup gain very close to the theoretical optimal acceleration. In fact, considering that this platform only incorporates 8 independent CPUs, the observed exception corresponds to a particular setup where the number of running threads was extended by using the HT technology. Nevertheless, all the observed results are entirely similar to those that were also obtained with dedicated SW frameworks [6, 10].

In the whole, it is observed that the slices independency allows a parallel execution with little data sharing between the several threads, thus minimizing the inherent segmentation and scheduling overheads. However, since the H.264 standard limits this frame division to a maximum of 16 slices, other

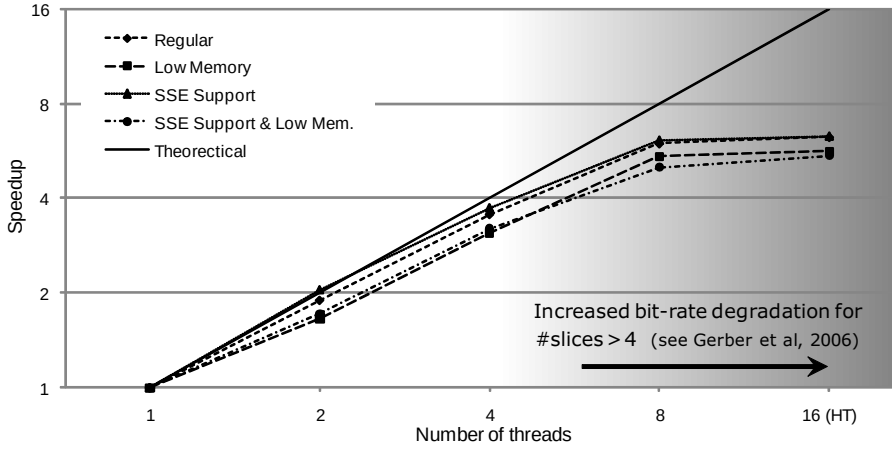


Figure 28.8 Provided speedup in Intel platform using *slice-level* parallelism.

levels of parallelization will have to be applied in order to avoid this constraint and allow greater levels of scalability.

28.4.3 Exploiting macroblock-level parallelism

As it was observed by Gerber *et al.* [10], the data independency that is achieved by dividing the frame into several slices often introduces a negative impact on the amount of spatial prediction that is exploited within a frame, with a consequent decrease of the resulting encoding efficiency. A direct consequence of such effect is a natural increase of the output bit-rate and a subsequent decrease of the resulting video quality, as a result of the application of the output buffer control mechanism. According to [10], such effect is particularly observed as soon as the number of slices is greater than 4 (see Fig. 18.6 of [10]). Hence, not only is the maximum number of independent slices low, but its increase gives rise to a consequent decrease of the encoding efficiency (see shaded region in Fig. 28.8).

To circumvent such undesirable effect, the added level of parallelism at macroblock-level (see Fig. 28.5(a)) that is also offered by the presented framework allows the several cores from the same group to cooperate in the concurrent processing of independent MBs, in order to further accelerate the encoding of the same slice without sacrificing the resulting bit-rate and encoding quality.

In the presented evaluation of this topology, this extra level of concurrency was implemented with nested threads. As soon as the slices are assigned to the primary threads, they are responsible to create other threads, in order to form *teams of threads*. Then, the set of independent MBs within each slice

Table 28.6 Provided speedup using *slice* and *macroblock* parallelism levels, by considering a maximum of 4 slices (to avoid bit-rate degradation [10]).

			Number of concurrent **Macroblocks** within each Slice			
			1	2	4	8
Intel™ E5530 (8 cores)	Number of concurrent **Slices**	1	1.0^{1}	1.7^{2}	3.0^{4}	4.4^{8}
		2	1.9^{2}	3.3^{4}	5.0^{8}	4.6^{16}
		4	3.6^{4}	5.3^{8}	6.3^{16}	–
AMD™ 8384 (32 cores)	Number of concurrent **Slices**	1	1.0^{1}	1.6^{2}	2.5^{4}	3.2^{8}
		2	2.3^{2}	4.1^{4}	3.7^{8}	4.5^{16}
		4	3.7^{4}	4.6^{8}	5.3^{16}	6.8^{32}

NOTES:
(1) - Number of concurrent threads represented in superscript, above the speedup value;
(2) - Intel's configurations using 16 threads make use of Hyper-Threading technology.

is distributed among the remaining available cores in each team. At the end, entropy encoding is performed separately and at *slice level*, to avoid memory bottlenecks and the usage of shared data.

The results presented in Table 28.6 illustrate the speedup levels that can be obtained either by an isolated or mutual exploitation of the *slice* and *macroblock* parallelism levels. Contrasting with the close-to-optimal results that were obtained with the *slice-level* model, the speedup values that are provided with an exclusive exploitation of the *macroblock-level* (Slice = 1) are somewhat more modest, achieving a maximum value of about 4.4 and 3.2 when using 8 threads in Intel's platform and 32 threads in AMD's platform, respectively. Data sharing between the several cores of these multi-processors is the main reason for this limitation, leading to a memory bottleneck and higher latency times.

An evaluation of the conceived capability to enhance the scalability of the parallel framework is also presented in Table 28.6. The simultaneous exploitation of *slice* and *macroblock* parallelism levels was assessed by using a set of configurations characterized by a different (but fixed) amount of threads: 1, 2, 4, 8, ... When compared with the previous results, it can be observed that the *macroblock* parallelism level, that is now possible to exploit as a complement to the *slice-level* parallelism, provides an extra speedup space beyond the *slice-level* baseline. Contrary to other highly specific and dedicated SW implementations [6, 10, 5, 18], such capability is now easily exploited by simply parameterizing the presented SW framework.

28.5 CONCLUDING REMARKS

A new open-software framework to implement parallel H.264/AVC encoders was presented in this chapter. Such framework significantly eases the exploita-

tion of the parallel processing capabilities offered by current homogeneous and heterogeneous multi-core architectures, as an efficient means to increase the resulting encoding performance.

Three different *functional-level* topologies are supported and easily parameterized by the presented SW framework: pipeline, straight parallel and mixed. In what concerns the exploitation of *data-level* parallelism, two models are particularly supported, besides the trivial frame-level partition model: slice level and macroblock level.

Despite the limitations imposed by the H.264 standard, *slice-level* parallelism proved to be the most efficient approach, with speedup gains quite close to the theoretical maximum. On the other hand, the *macroblock-level* parallelism model has shown to offer an extra and quite viable speedup margin that can be exploited as a complement to the *slice-level* parallelism, in order to improve the scalability of the parallel implementation, in particular, when the number of available cores is greater than 16.

Acknowledgments

This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

REFERENCES

1. M. Ali Ben Ayed, A. Samet, and N. Masmoudi. H.264/AVC prediction modules complexity analysis. *Wiley InterScience on European Transactions on Telecommunications*, 18:169–177, Aug. 2007.
2. A. Azevedo, B.H.H. Juurlink, C.H. Meenderinck, A. Terechko, J. Hoogerbrugge, M. Alvarez, A. Ramirez, and M. Valero. A highly scalable parallel implementation of H.264. *Transactions on High-Performance Embedded Architectures and Compilers (HiPEAC)*, Sep. 2009.
3. Jongwoo Bae, Youngsung Soh, Daewon Kim, and Myungho Lee. A programmable multi-format video decoder. In *Proceedings of International Conference on Intelligent Computation Technology and Automation (ICICTA'09)*, volume 4, pages 25–28, 2009.
4. Wei-Nien Chen and Hsueh-Ming Hang. H.264/AVC motion estimation implmentation on Compute Unified Device Architecture (CUDA). In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME'2008)*, pages 697–700, 2008 2008.
5. Y.-K. Chen, E. Q. Li, X. Zhou, and S. L. Ge. Implementation of H.264 encoder and decoder on personal computers. *Journal of Visual Communications and Image Representations*, 17(2):509–532, Apr 2006.
6. Y.-K. Chen, X. Tian, S. Ge, and M. Girkar. Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures. In *Proceedings of*

International Parallel and Distributed Processing Symposium, pages 63–72, Apr 2004.

7. Ngai-Man Cheung, Xiaopeng Fan, Oscar C. Au, and Man-Cheung Kung. Video coding on multicore graphics processors. *IEEE Signal Processing Magazine*, 27(2):79–89, March 2010.
8. Tiago Dias, Nuno Roma, and Leonel Sousa. H.264/AVC framework for multi-core embedded video encoders. In *Proceedings of International Symposium on System-on-Chip (SOC'2010)*, pages 89–92, September 2010.
9. Tiago Dias, Nuno Roma, Leonel Sousa, and Miguel Ribeiro. Reconfigurable architectures and processors for real-time video motion estimation. *Journal of Real-Time Image Processing*, 2(4):191–205, 2007.
10. Richard Gerber, Aart J. C. Bik, Kevin Smith, and Xinmin Tian. *The Software Optimization Cookbook*, chapter 18 - Case Study: Threading a Video Codec, pages 323–343. Intel Press, second edition, 2006.
11. Xun He, Xiangzhong Fang, Ci Wang, , and Satoshi Goto. Parallel HD encoding on CELL. In *Proceedings of International Symposium on Circuits and Systems (ISCAS2009)*, page 10651068, May 2009.
12. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.
13. S. Hiratsuka, S. Goto, and T. Ikenaga. An ultra-low complexity motion estimation algorithm and its implementation of specific processor. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS'2006)*, 2006.
14. *JM H.264/AVC Reference Software*. http://iphome.hhi.de/suehring/tml/, 2011.
15. S. Kim and M. Sunwoo. ASIP approach for implementation of H.264/AVC. *Journal of Signal Processing Systems*, 50(1):53–67, Jan. 2008.
16. H. Mizosoe, D. Yoshida, and T. Nakamura. A single chip H.264/AVC HDTV encoder/decoder/transcoder system LSI. *IEEE Transactions on Consumer Electronics*, 53(2):630–635, May 2007.
17. Svetislav Momcilovic and Leonel Sousa. Modeling and evaluating non-shared memory CELL/BE type multi-core architectures for local image and video processing. *The Journal of Signal Processing Systems*, 62(3):301–318, March 2010.
18. A. Rodríguez, A. González, and M.P. Malumbres. Hierarchical parallelization of an H.264/AVC video encoder. In *International Conference on Parallel Computing in Electrical Engineering*, pages 363–368, 2006.
19. Xinmin Tian, Yen-Kuang Chen, M. Girkar, S. Ge, R. Lienhart, and S. Shah. Exploring the use of hyper-threading technology for multimedia applications with Intel OpenMP compiler. In *Proceedings of International Parallel and Distributed Processing Symposium*, pages 1–8, April 2003.
20. Sung-Wen Wang, Ya-Ting Yang, Chia-Ying Li, Yi-Shin Tung, and Ja-Ling Wu. The optimization of H.264/AVC baseline decoder on low-cost TriMedia DSP processor. In *Proceedings of SPIE*, volume 5558, pages 524–535, 2004.
21. T. Wiegand, Gary J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *Transactions on Circuits and Systems for Video Technology*, 13:560–576, Jul. 2003.