Evaluation of data-parallel splitting approaches for H.264 decoding

Florian H. Seitner
Vienna University of Technology
seitner@ims.tuwien.ac.at

Michael Bleyer
Vienna University of Technology
bleyer@ims.tuwien.ac.at

Ralf M. Schreier
Vienna University of Technology
schreier@ims.tuwien.ac.at

Margrit Gelautz
Vienna University of Technology
gelautz@ims.tuwien.ac.at

ABSTRACT

The high computational demands of the H.264 decoding process pose serious challenges on current processor architectures. A natural way to tackle this problem is the use of multi-core systems. The contribution of this paper lies in a systematic overview and performance evaluation of parallel video decoding approaches. Our study investigates six methods for accomplishing data-parallel splitting in strongly resource-restricted environments inherent to mobile devices. These methods are compared against each other in terms of run-time complexity, core usage, inter-communication and bus transfers. We present benchmark results using different numbers of processor cores. Our results shall aid in finding a splitting strategy that is best suited for the targeted hardware-architecture.

Categories and Subject Descriptors

I.4 [**Image Processing and Computer Vision**]: Compression (Coding)

Keywords

Video, decoding, H.264/AVC, parallel, embedded architectures.

1. INTRODUCTION

The H.264 video standard [4] is currently used in a wide range of video-related areas such as video content distribution and television broadcasting. Compared to preceding standards such as MPEG-2 and MPEG-4 SP/ASP, improved coding efficiency could be reached by introducing more advanced pixel processing algorithms (e.g. quarter-pixel motion estimation) as well as by the use of more sophisticated algorithms for predicting syntax elements from neighbouring macroblocks (e.g. context-adaptive VLC). These new coding tools result in significantly increased CPU and memory loads required for decoding a video stream. In environments of limited processing power such as embedded systems, the high computational demands pose a challenge for practical H.264 implementations. Multi-core systems provide an elegant and power-efficient solution to overcome these performance limitations.

However, the efficient distribution of the H.264 video algorithm among multiple processing units is a non-trivial task. For using the available processing resources efficiently, an equally balanced distribution of the decoder onto the hardware units must be found. The system designer has to consider data dependency issues as well as inter-communication and synchronization between the processing units. Furthermore, the resource limitations in an embedded environment such as low computational power, small-sized on-chip memories and low bus bandwidth require an efficient software design. A parallel decoder approach for mobile systems must be able to work under these resource restrictions.

In previous work, various approaches for parallelizing the H.264 decoding process have been introduced. Van der Tol et al. [14] have investigated methods for mapping the H.264 decoding process onto multiple cores. Functional splitting of an H.264 decoder and the use of macroblock pipelining at thread-level have been demonstrated in [2, 3, 10]. Zhao et al. [16] exploit frame parallelism in the Wavefront technique. A hierarchical approach working at group-of-picture and frame level is demonstrated in [9]. The scalability of H.264 for a data-parallel decoding approach operating on the macroblock-level and on multiple frames in parallel has been investigated by Meenderinck et al. [6]. The same study introduces an efficient technique for H.264 frame-level parallelization, the 3D-Wave strategy.

These papers primarily focus on parallelization in terms of algorithmic scalability. Upper limits on the number of processors and frames processed in parallel are given. However, the memory-restrictions of embedded environments make these approaches hardly usable for mobile and embedded architectures. More resource-efficient H.264 splitting approaches have been introduced in [12, 13, 15]. The focus of these authors has been put on efficient decoder implementations for embedded architectures.

The aim of this work is to evaluate the behaviour of differ-

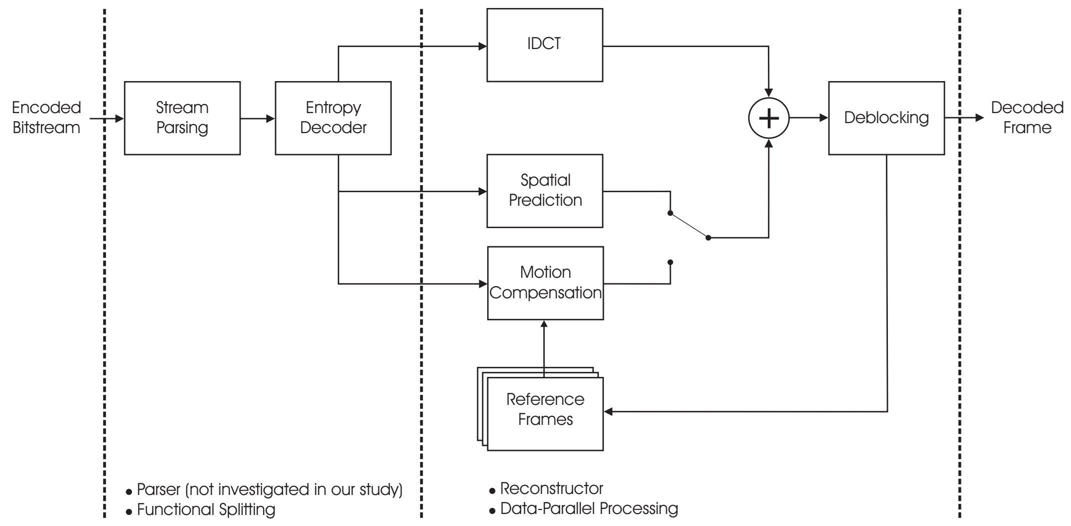


Figure 1: The H.264 decoding process.

ent parallel decoding approaches in terms of run-time complexity, efficient core usage, data transfers and buffer sizes. This information is of prime importance when designing a resource-efficient multi-core decoder. The information can be exploited for choosing the parallel approach which performs best under certain hardware restrictions. Our study addresses the following issues:

- We provide a detailed overview of six different data-parallel splitting methods for H.264. These methods represent the main approaches addressed in the literature and work under resource-restricted environments such as those of mobile devices.
- We introduce a high-level simulation approach for modelling parallel systems that is used for comparing these six approaches.
- We investigate the approaches' run-time issues in a systematic way.
- We compare these approaches against each other and evaluate the impact of raising core counts on the resource requirements using different test sequences.

The remainder of the paper is organized as follows. Section 2 focuses on the fundamental differences between functional and data-parallel splitting approaches. Additionally, it describes the H.264 decoding process and the algorithmic dependencies that counteract the parallelization. Section 3 gives a detailed overview of the six evaluated parallelization approaches. We provide a detailed analysis and describe our motivation for including them in our survey. In Section 4, we describe our evaluation methodology. Results are provided in Section 5. Final conclusions and an outlook are given in Section 6.

2. PARALLEL H.264 DECODING

2.1 Functional and data-parallel splittings

The memory restrictions of most embedded video coding architectures require the use of small data packages and typically result in a macroblock-based decoding system. In functional partitioned decoding systems, the decoding tasks such as parsing, motion compensation or deblocking are assigned to individual processing cores, typically one task per each processing unit. Macroblocks are processed by one processor after the other. The multiple processors allow starting the next macroblock's decoding tasks before computation of the current macroblock has finished.

This splitting method has the advantage that each processing unit can be optimized for a certain task (e.g. by adding task-specific hardware extensions) and minimal-sized instruction caches. In contrast to data-dependent parallelization, also strongly sequential tasks can be accelerated by this strategy. The disadvantages are an unequal workload balancing and high transfer rates for inter-communication.

As opposed to functional splitting methods, data-parallel systems do not distribute the functions, but the macroblocks among multiple processing units. For efficient parallelization, the macroblocks' core assignment algorithm has to address the following issues:

- The data-dependencies between different cores must be minimized and data-locality must be exploited (i.e. supporting of caching strategies).
- The macroblock distribution onto the processing cores must achieve an equal workload balancing.
- Generic macroblock core assignment for different frame sizes must be possible.

Scalability in parallel systems requires minimal data-dependency between the cores. By grouping the macroblocks as described in [14], a compromise between small memory size and data-dependencies can be reached. For supporting caching strategies at a more global scale, the groups of macroblocks assigned to one core must be aligned closely together in each frame. By introducing a centralized and constant macroblock assignment for each frame, this global caching issue can be addressed efficiently. Additionally, a deterministic macroblock assignment allows the parsed macroblocks to be written directly to the input FIFOs of the corresponding reconstructing cores.

In this work, we focus solely on approaches based on data-parallelization. For higher core counts, this parallelization

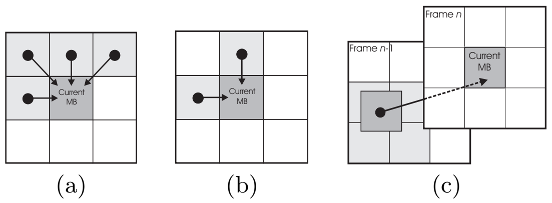


Figure 2: Macroblock dependencies in the H.264 decoder. Arrows mean that the macroblock at the origin of the arrow needs to be processed before decoding the other macroblock. (a) Intra-prediction dependency. (b) Deblocking filter dependency. (c) Inter-prediction dependency.

paradigm offers better scalability and a more balanced workload than functional splitting approaches [14].

2.2 The H.264 decoder

Figure 1 illustrates the principle of the H.264 decoding process. The decoder can be divided into two fundamentally different parts, the parser and the reconstructor.

The parser includes context calculations for the context-adaptive part of the entropy decoding process, the low-level entropy decoding functions (binary-arithmetic or variable length coding) and the reconstruction of the macroblock's syntax elements such as motion vectors or residual information. This decoder part is highly sequential with short feedback loops at bit- and syntax element-levels. Furthermore, the conditional calculations of the entropy coder's context information require short branch executions of the executing processor and efficient data access. This makes data-parallel processing hardly usable in these decoding stages and functional splitting is preferable. Our work therefore focuses on the reconstructor module of the H.264 decoder, which provides a wide range of possibilities for exploiting data-parallelization. In our simulations, we assume that stalls cannot be caused by the parser. In other words, an idealized parser always has those macroblocks available that are currently needed in the reconstructor module.

In the reconstruction part, the H.264 decoder uses the parsed syntax elements to generate spatial (=intra) or temporal (=inter) predictions for each macroblock. The residual coefficients are inverse transformed (IDCT) and added to the macroblock's prediction. Finally, an adaptive deblocking filter is applied to the outer and inner edges of the 16 4×4 pixel sub-blocks of a macroblock in order to remove blocking artefacts.

2.3 Macroblock dependencies

Data-parallel splitting of the decoder's reconstruction module is challenging due to dependencies between spatial as well as temporal neighbouring macroblocks. These dependencies originate from three sources illustrated in Figure 2 and described as follows.

Firstly, in progressive frame coding, the intra-prediction uses unfiltered pixel information from up to four spatially neighbouring macroblocks to predict the current macroblock. These dependencies are depicted in Figure 2(a). In general, it is a good option to gather the current macroblock and its reference macroblocks on the same CPU to avoid expensive inter-processor communication for resolving this dependency. For an efficient parallelization, this dependency must be addressed carefully.

Secondly, the deblocking filter imposes additional spatial dependencies. For filtering the outer edges of the current macroblock, up to four pixel rows / columns from the upper and left neighbouring macroblocks are used as filter input. These macroblock dependencies are visualized in Figure 2(b). An efficient parallelization method will focus on avoiding that these dependencies have to be resolved across individual processors.

The third and final macroblock dependency arises from the inter-prediction. The inter-prediction reads reference data from macroblocks of previously decoded frames. Obviously, it is required that processing of these reference macroblocks has already been completed before they can be used for inter-prediction of the current macroblock. This results in the temporal dependency depicted in Figure 2(c). In fact, the current macroblock can depend on a rather large amount of reference macroblocks. H.264 allows splitting of the current macroblock into small sub-blocks for each of which a separate motion vector is computed. In P-slices, each inter-coded macroblock can contain up to 16 motion vectors and point to one reference frame. For bi-directional predicted macroblocks in B-slices, a maximum of 32 motion vectors and two reference frames is possible.

3. EVALUATED METHODS

The approaches considered in this evaluation represent the main groups of data-parallel approaches addressed in the literature. All of them work under the restrictions of typical embedded architectures. These restrictions include low system memory and buffer sizes on the hardware side as well as thread-less and application-dedicated run-time environments on the OS side.

Our evaluation study compares the performance of six different approaches for accomplishing data-parallel splitting of the decoder's reconstructor module. These are (1) a Single-row approach, (2) a Multi-column approach, (3, 4) a blocking and a non-blocking version of a Slice-parallel method, (5) a dynamic version of the non-blocking Slice-parallel approach as well as (6) a Diagonal technique. Details on the benchmarked approaches are provided in the following.

3.1 Single-row approach

The first splitting strategy investigated in our study distributes horizontal lines of macroblocks among different processors as shown in Figure 3. More formally, let N be the number of processors. Processor $i \in \{0, \cdots, N-1\}$ is then responsible for decoding the yth row of macroblocks if $y \bmod N = i$.

To illustrate the Single-row (SR) approach, we give an example with two processors on an image divided into 8×8 macroblocks in Figure 4. In this example as well as in the following ones of Figures 7, 9, 10 and 12, we assume that it takes a constant value of 1 unit of time to process each macroblock. It is, however, important to notice that this is a coarse oversimplification. In real video streams, there are large variations in the processing times of individual macroblocks, which makes it difficult to evaluate the goodness of a parallelization approach. In Figure 4, only processor 1 is able to decode macroblocks at time $t = 2$, since all other macroblocks are blocked as a consequence of the dependencies illustrated in Figure 2. After the first two macroblocks of row 1 have been computed, the second core can start

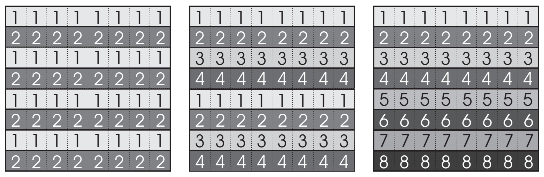

(2 Cores) (4 Cores) (8 Cores)

Figure 3: The Single-row splitting approach. The assignment of processors to macroblocks is shown.

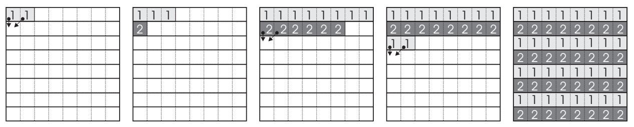

($t = 2$) ($t = 3$) ($t = 8$) ($t = 10$) ($t = 34$)

Figure 4: Example of the Single-row splitting approach used with two cores. Processed macroblocks are shown at different instances of time t. It takes a constant value of 1 unit of time to process a macroblock.

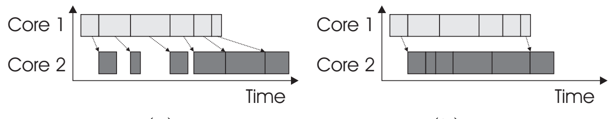
processing the first macroblock of the second row, since the dependencies for this macroblock are now resolved ($t = 3$). The next interesting event occurs at $t = 8$ when CPU 1 has finished the computation of the first row. Macroblocks of the second row have already been computed and therefore CPU 1 can start decoding macroblocks of row 3 that are dependent on their upper neighbours. At time $t = 10$ we obtain the same situation as at $t = 2$, where the first core unlocks the second one. Finally, the whole frame has been decoded at $t = 34$.

The advantage of the Single-row approach lies in its simplicity. It is very easy to split the frame among the individual processors. There is only a small start delay after which all processors can effectively work. The potential downside of this approach is that there are many dependencies that need to be resolved across processor assignment borders. This has played no role in our example where we have assumed constant processing time for each macroblock. It, however, will be noticable for real videos streams that contain macroblocks of considerably different run-time characteristics. In fact, each macroblock processed by core i depends on its upper neighbours that are processed by a different core $i - 1$. If processor $i - 1$ fails to deliver these macroblocks at the right time, this will immediately produce stalls at core i. This behaviour is shown in Figure 5(a). On the other hand, this strong coupling of CPUs can potentially lead to low buffer requirements.

3.2 Multi-column approach

The second splitting method of our study divides the frame into equally-sized columns as shown in Figure 6. Let w denote the width of a multi-column that is typically derived by dividing the number of macroblocks in a row by the number of processors. Processor i is then responsible for decoding a macroblock of the xth column if $iw \leq x < (i+1)w$. A similar method to partition the image has recently been proposed for the H.264 encoder in [13].

Figure 7 simulates the Multi-column (MC) approach using two processors. CPU 1 thereby starts processing the first row of macroblocks until it hits the border to the macroblocks assigned to processor 2 ($t = 4$). Since the dependency for the leftmost macroblock of CPU 2 is now resolved, processor 2 can finish decoding its first macroblock at $t = 5$. We obtain a similar situation at $t = 8$. The dependencies of the leftmost macroblock of the second row have been resolved, and CPU 2 can therefore continue its work. Decoding of the frame is finally completed at $t = 36$.



(a) (b)

Figure 5: The number of inter-processor dependencies is crucial for the overall performance of the multi-core system. Rectangles represent macroblocks. A macroblock's width indicates the required processing time. Arrows between two macroblocks mean that processing of the macroblock which the arrow points to can only start after the other macroblock has been decoded. (a) A large number of inter-processor dependencies slows down the system. (b) Due to the low amount of inter-processor dependencies, different running times of individual macroblocks become averaged out. This should improve the overall performance.

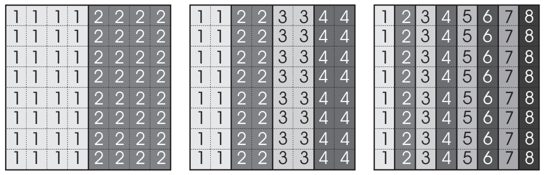

(2 Cores) (4 Cores) (8 Cores)

Figure 6: The Multi-column splitting approach.

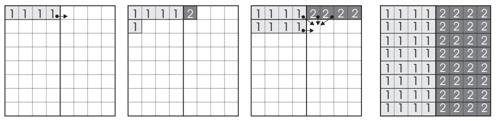

($t = 4$) ($t = 5$) ($t = 8$) ($t = 36$)

Figure 7: Example of the Multi-column splitting approach.

The basic idea behind using the Multi-column approach is to obtain a looser coupling of processor dependencies. In fact, the processor assignment borders are significantly reduced in comparison to the Single-row approach. One processor has to wait for the results of another one only at the boundary of its multi-column. Within the multi-column, macroblock dependencies can be resolved on the same processor. This should lead to reduced inter-processor dependencies and could therefore improve the overall run-time behaviour of the multi-core system as is depicted in Figure 5(b).

3.3 Slice-parallel approach

As a third splitting method, we investigate a 90-degree rotated version of the Multi-column approach that divides the frame into even-sized rows. This method is depicted in Figure 8. Formally spoken, let h denote the height of a multi-row. A macroblock of the yth row is then assigned to CPU i if $ih \leq y < (i + 1)h$.

The run-time behaviour of the Slice-parallel (SP) approach

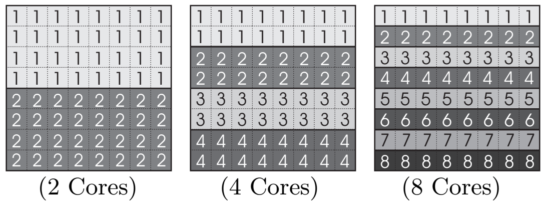


Figure 8: The Slice-parallel splitting approach.

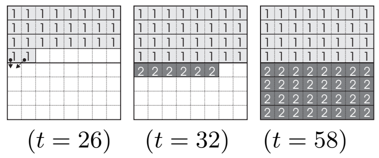


Figure 9: Example of the Slice-parallel approach in the blocking version.

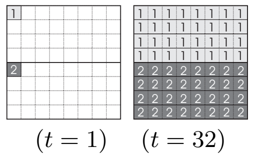


Figure 10: Example of the Slice-parallel approach in the non-blocking version.

is illustrated in Figure 9. Here, CPU 2 has to wait for a relatively long time ($t = 26$) until the dependencies for its first assigned macroblock are resolved. While the first processor can complete its work on the current frame at $t = 32$, it still takes 26 units of time until the second CPU finishes processing the remainder of the frame at $t = 58$. In the following, we refer to this approach as the blocking Slice-parallel technique.

The incorporation of the Slice-parallel approach into our benchmark is also motivated by recent work [7] that has presented a non-blocking encoder version of this method. The authors encode their video streams so that slice borders coincide with horizontal lines in the frames. Since neither dependencies introduced by intra-prediction nor dependencies introduced by the deblocking filter occur across slice borders, the multi-rows can be processed completely independent from each other.

Obviously, this non-blocking SP approach (NBSP) requires having full control over the encoder, which will not be the case for many applications. This downside should be considered when interpreting the results for this method in our benchmark. For completeness, we also give an example of this approach in Figure 10. Here, both CPUs can start processing their assigned macroblocks immediately ($t = 1$) and finish the decoding of the complete frame at $t = 32$.

3.4 Rotating slice-parallel approach

Content-dependent differences in the decoding complexity of the slices will result in an unequal workload of each core [11]. For example, static and low textured frame regions will use coarse macroblock partitions and few residual information. This requires less processing power than for moving and strongly textured regions. If we assign a slice with low complexity constantly to the same core, a drift between this core and the cores working on more complex slices will occur on the long term in the non-blocking NBSP approach. This will result in low system usage and high buffer size requirements for compensating the differences in the workload.

For investigating the impact of the sequence content on the NBSP approach's run-time behaviour, we introduce a modified version of this approach (NBSPr approach). It supports a rotating core assignment depending on the frame number. For example, in a two-core splitting, the first processor would process the upper image part in the first frame, the bottom part in the second frame, then again the upper part and so on. In contrast to the NBSP approach with static assignment, on average, this rotating approach should reach a more balanced system workload.

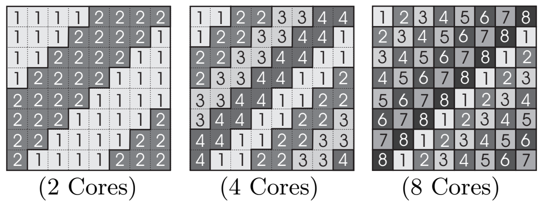


Figure 11: The Diagonal splitting approach.

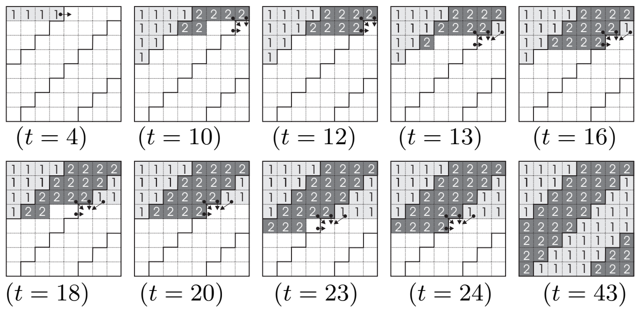


Figure 12: Example of the Diagonal approach.

3.5 Diagonal approach

As a final approach, we include the splitting method illustrated in Figure 11 into our evaluation. This processor assignment is obtained by dividing the first line of macroblocks into equally-sized columns. The assignments for the subsequent lines are then derived by left-shifting the macroblock assignments of the line above. This procedure leads to diagonal patterns.

Figure 12 gives an example of the diagonal (DG) approach using two processors. Here, the second CPU stalls until its dependencies become resolved by CPU 1 at $t = 4$. The

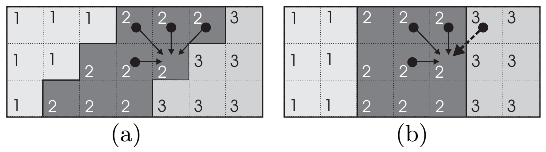


Figure 13: Processor dependencies in the Diagonal and Multi-column approaches. (a) In the Diagonal method, dependencies for CPU 2 originate solely from macroblocks assigned to CPU 1. CPU 2 therefore never has to wait for CPU 3. (b) In the Multi-column approach, macroblocks assigned to CPU 2 are also dependent on processor 3 as indicated by the dotted arrow.

first CPU completes computation of its first image partition at $t = 10$. Unfortunately, it cannot directly start processing the second partition, but has to wait for CPU 2 to resolve dependencies until $t = 12$. The following images ($t = \{13, 16, 18, 20, 23, 24\}$) show situations where the first CPU has to wait for macroblocks decoded by processor 2. For legibility, we do not show subsequent states where one processor blocks the other one, but directly proceed to the final result derived at $t = 43$.

Inclusion of the Diagonal approach into our evaluation is motivated by the macroblock partitioning approaches proposed in [14]. The Diagonal splitting method is regarded as an approach that “respects” the dependency patterns spanned by the intra-prediction and the deblocking filter. (Dependencies are shown in Figure 2). We illustrate the idea behind the Diagonal splitting method in Figure 13. The figure compares the inter-processor dependencies introduced by Diagonal and Multi-column splitting techniques. The Diagonal method thereby only shows dependencies on macroblocks from its left neighbouring processor, which is in contrast to the Multi-column method that contains dependencies on macroblocks of both neighbouring CPUs. These reduced inter-processor dependencies could lead to an improved run-time behaviour of the multi-core system.

4. EVALUATION METHODOLOGY

4.1 Test sequences

For comparing the described parallelization approaches, we use five standard image sequences listed in Table 1. In compiling this test set, we aim to reach a high diversity in the test sequences’ contents and on covering the whole complexity range of typical H.264 sequences at CIF resolution. For encoding the test streams, we apply the most commonly used coding tools supported in the main profile of the H.264 standard. With the exception of interlaced coding, all main profile coding tools such as I-, P- and B-slices as well as weighted prediction are switched on. The test sequences are encoded using the JM12.2 encoder [5] with parameters chosen as follows: H.264 main profile, GOP size 14 frames, CIF, IPB, VLC, deblocking active, all prediction modes allowed, SR +/–16 pixels, 5 reference frames.

To avoid impacts of the encoder rate control algorithm as well as coding quality effects on the decoder run-time, all sequences are coded with constant quantization parameters at an average SNR of 35 dB. We therefore adjust the values of parameters QP_I, QP_P and QP_B that are responsible for quantization of I-, P- and B-frames so that each individual frame of the sequence exhibits an SNR of approximately 35dB. The corresponding settings for the quantization parameters as well as resulting bitrates are found in Table 1.

4.2 Simulation environment

For investigating the complex dynamic behaviour of the benchmarked parallelization strategies, we have created a high-level simulation environment. This simulator is motivated by the work of Bondarev et al. [1] and has also been used in [11, 12].

In the simulation environment, it is possible to describe different splitting methods on arbitrary multi-core hardware architectures and to simulate their run-time behaviour when decoding a specific H.264 stream. The principle of the simulator is shown in Figure 14 and described as follows. The user defines the targeted hardware architecture. This includes properties of the processing units (such as their number), memories and the data communication interfaces. In addition, disjunct sets of macroblocks that shall be processed by the individual CPUs are defined. To compute the overall run-time as well as other data evaluated in our benchmark, we make use of profiling data collected for the investigated video stream. (More information on generating the profiling data is given in Section 4.3.) The most important profiling data is thereby the amount of processing time required for decoding a specific macroblock. In the figure, the required processing time is represented by the different widths of individual macroblocks (rectangles). Arrows between macroblocks denote dependencies. The dotted arrows represent macroblock dependencies that have to be resolved across different CPUs according to the illustrated processor assignments. Our simulator accounts for these inter-processor dependencies. It stalls a processor until the dependency for the currently processed macroblock is resolved. Roughly described, our simulator computes the overall processing time by taking into account the profiled run-times of individual macroblocks plus the time of stalls that occur within the simulation of the benchmarked splitting method. Apart from the computational complexity, additional resources such as data transfers or buffer requirements are modelled in the simulation environment which allows detailed design space explorations.

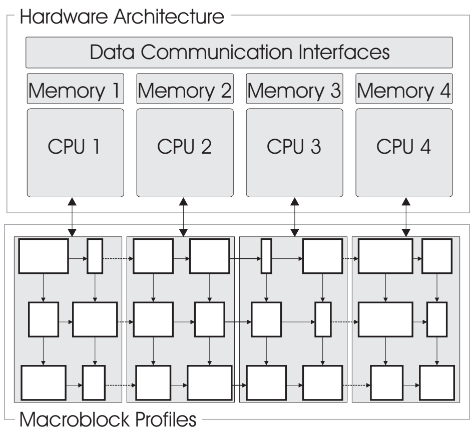


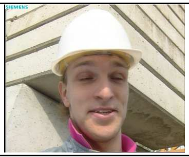
Figure 14: Principle of the simulator. Explanation is given in the text.

The advantages of a high-level simulation approach lies in fast and flexible modelling of different hardware architectures. This avoids a time-consuming hardware realization. Moreover, the complex and error-prone task of implementing each parallelization approach is eliminated.

4.3 Macroblock profiling

The macroblock profiles required for our high-level simulator are derived from a single-core hardware simulator for the very long instruction word (VLIW) multimedia processor CHILI [8]. The hardware simulator’s accuracy has been verified with an ASIC implementation of this processor. The CHILI is a RISC processor. It can process four instructions in parallel which can be any combination of 32-bit arithmetic instructions and load-store operations. For parallel pixel operations, 16-bit SIMD instructions are provided. The pro-

Table 1: Five test sequences at CIF resolution and with normalization to 35 dB. Rows 2 and 3 show the PSNR values used for normalizing the sequences and the resulting bitrates. The quantization values used for the coding of I-, P- and B-frames are given in rows 4 to 6.

Sequence	**Foreman**	**Bus**	**Flowergarden**	**Mobile**	**Barcelona**
					
PSNR [dB]	35	35	35	35	35
Bitrate@25Hz [kbit/s]	250	1000	1350	1750	2250
QP_I	33	29	30	29	28
QP_P	31	28	28	27	26
QP_B	31	28	28	26	25

cessor can be programmed in C/C++ with intrinsic SIMD functions as well as in low-level assembly language for time-critical routines.

In order to obtain the profiling data, the hardware simulator for the VLIW media processor and its data memory system (DMS) are extended to enable function profiling on a macroblock level. Hence, the hardware simulator provides very detailed records of program execution from which the call graph as well as individual execution times for each function call can be extracted. The exact execution time for each macroblock is measured by accumulating the function cycle-counts of all sub-functions for each functional block.

Our profiles are based on a commercial H.264/AVC main profile decoder for embedded architectures that has been optimized in terms of memory usage and support of DMA transfers. In addition, the regular pixel-based processing functions of the decoder (e.g. interpolation, prediction) have been assembly optimized to make use of the SIMD processor commands. Macroblock-based run-time measurements are computed for all test sequences of Table 1.

5. RESULTS

5.1 Run-time complexity

Two major indicators for the efficiency of a multi-core decoding system are the decoder's run-time and the number of data-dependency stalls occurring during the decoding process. A low run-time indicates a high system decoding performance, while the number of stalls provides an estimate on how efficiently the system's computational resources are used.

In Figures 15 and 16, the decoder speed-up compared to the single-core version and the stall cycles are shown for the six investigated approaches. Additionally, Figure 17 provides run-time estimates. In all figures, it is observed that the SR method and MC method as well as the NBSP approach perform considerably better than the DG and the blocking SP approaches. Surprisingly, the SR approach performs best for most cases and even outperforms the NBSP approach for 2 and 4 cores. Only for 8 cores, the NBSP approach performs better.

Looking more into detail, we found that the constant core assignment to frame regions and the run-time differences between these regions result in an unbalanced workload distribution for the NBSP approach. According to our results, approaches working on a finer granularity level and alternating the cores over the frame more frequently (e.g. the SR approach) adapt better to local complexity peaks. With increasing number of cores the NBSP approach becomes more similar to the SR approach, with the advantage of fewer dependencies between the splitting partitions.

The strongest impact of the rotating order in the NBSPr approach can be observed for higher core counts. For 4 and 8 cores, the NBSPr outperforms the other approaches in most cases. For 2 cores, the stall cycles are already nearly optimal for the static NBSP approach. Here, the rotation even results in a small performance decrease for the Barcelona and Flowergarden sequence. For the Flowergarden sequence, the partition sizes and the decoding complexity varies significantly between the upper (sky) and lower (meadow) half of each frame and between P- and B-frames. These spatial as well as temporal complexity differences counteract the rotating assignment in this case. For the Barcelona sequence, the largest complexity differences between the slices is occurring in the bi-directional coded frames. The border regions of these frames contain mostly inter-coded macroblocks and the inner regions are mostly skipped macroblocks with low deblocking complexity.

Comparing only the encoder-independent approaches shows that the SR and the MC approaches have similar scalability. For the SR approaches, the different complexity between adjacent lines accumulates and results in a faster increase of the stall cycles for increasing core counts. The simulations show that the performance for the MC approach with 8 cores decreases strongly compared to the systems with fewer cores. On the one hand, this is caused by the number of columns which cannot be divided equally into 8 columns. On the other hand, the column width decreases with each additional core and results in a more frequent inter-
communication between the cores. However, for larger resolutions such as 720p or 1080p, which are multiples of 8, this effect should be negligible. The diagonal (DG) approach shows relatively poor performance, especially for higher core counts. For each additional core, the DG approach creates two additional partitions with dependencies on other cores' partitions. Adding a new core to the system raises the number of dependencies faster than for the other approaches and results in a higher number of stalls.

For the SP approach, the coarse horizontal splitting and

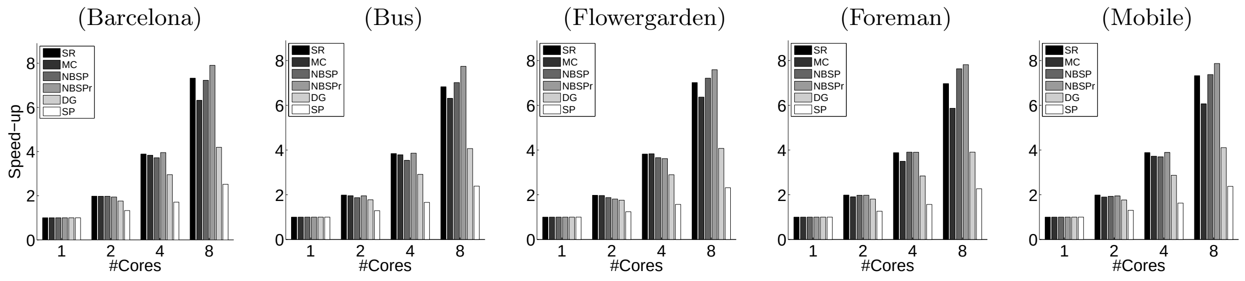


Figure 15: Speed-up in run-time. The speed increase for each parallelization approach in multiples of the single-core performance is shown for the five sequences considering 1, 2, 4 and 8 cores. The parallelization methods are: the Single-row method (SR), the Multi-column method (MC), the non-blocking versions of the Slice-parallel method (NBSP/NBSPr), the Diagonal method (DG) and the blocking version of the Slice-parallel method (SP).

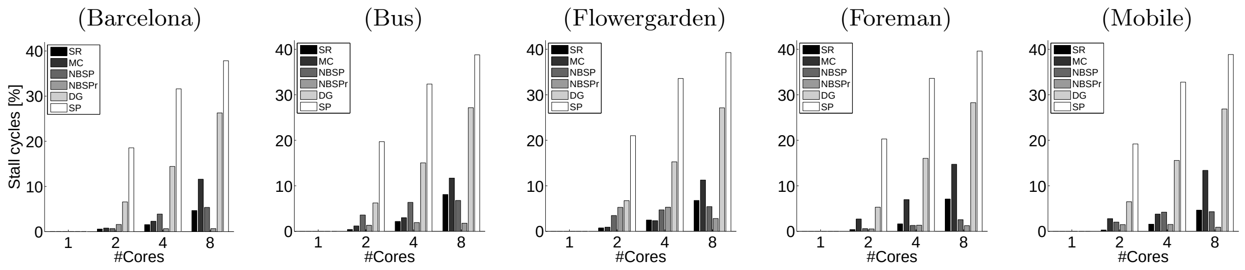


Figure 16: Stall cycles caused by data-dependencies between the cores. We plot the percentage of stall cycles in relation to the overall cycles spent for decoding a sequence.

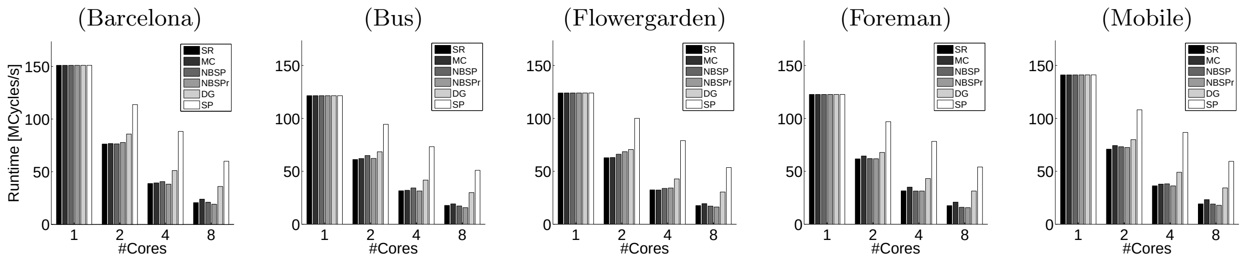


Figure 17: Average run-time per second for five test sequences. For the parallelization approaches, the system's run-time is shown for 1, 2, 4 and 8 cores. For the slice non-blocking approach, the core mapping was rotated according to the frame number to achieve an equal core usage (NBSPr approach).

the dependencies between each partition and its upper partitions results in high stall counts and the worst run-time of all approaches.

5.2 Inter-communication

Memory transfers to and from the external DRAM and between the cores' local memories are expensive in terms of power consumption and transfer times. In a video decoding system, these transfers are strongly influenced by the core inter-communication and the loading of reference data and deblocking pixels. Depending on the size of the cores' local memories and how efficiently the parallelization approach exploits locality, macroblock information can be kept on a core and reused for the decoding of future macroblocks. In our work, we assume that one macroblock line of intra-prediction and deblocking data can be kept in the core memory of each core. The resulting transfer rates for a CIF resolution video are shown in Figure 18. Note, that for higher resolutions such as 720p or 1080p caching of a complete line will not be possible. The memory increases will be more extreme.

For the MC approach, few data has to be exchanged between the cores and efficient data reuse is possible. This results in the lowest transfer rate of all encoder-independent approaches and even outperforms the blocking SP approach. Increasing the core counts only results in a slight transfer increase for the MC and the SP approaches. For the NBSP and the NBSPr approaches, the data communication even decreases for higher core counts since the number of independent frame partitions increases. However, this results in blocking artefacts at slice borders and a less efficient prediction.

The transfer rates for the DG approach are slightly worse than the MC and the SP approaches for 2 cores. The two additional partitions for each additional core result in new data dependencies to neighbouring partitions and a stronger increase of the data transfers when raising the number of cores.

The SR approach only exploits data locality between horizontal neighbouring macroblocks. The fine horizontal splitting into macroblock lines results in the highest transfer rate of all approaches and represents the worst case for horizontal splitting approaches in terms of memory transfers. Note that for two or more cores the transfer rate of the SR approach is constant. The alternating processing order of the macroblock lines requires the transfer of vertical reference data between the cores.

5.3 System buffer requirements

For multiple-core decoding, the reconstruction cores are working on different macroblock locations in the frame. All parsed macroblocks must be kept in a buffer until they are processed by a reconstructor core. The size of this buffer depends on the macroblocks' processing order which is strongly influenced by the frames' partition sizes and shapes.

Figure 19 shows the maximum number of macroblocks that must be kept in the parser output buffer to avoid write stalls at the parser side. The two slice-based approaches clearly have the highest requirements on buffering macroblocks. For the blocking SP as well as for the NBSP approach, a strong increase for higher core counts can be seen.

In the blocking version, the first slice of a frame can only have data-dependencies to previous frames due to the inter-prediction. The other slices have to wait until the upper most slice is finished. After finishing the first slice, the core moves to the next frame's first slice. This requires a minimal buffer size $B_{min} = N * (C - 1)/C$. Here N is the number of macroblocks in a frame and C the number of processors. Since the first slice works as a synchronization mechanism for the other slices, the average decoding time for a frame and therefore the buffer requirements (Figure 19) only vary moderately. For the NBSP and the NBSPr approaches, the time differences in the slices' decoding time have a greater impact since no dependencies between the slices of a frame exist. Hence, only the availability of inter-prediction dependencies keeps the fastest core from running away which would result in a buffer size increase. We can see in Figure 19 that the buffer requirement is very dependent on the sequences' content. Vertical movement in the sequences and the number of inter-coded macroblocks and reference frames influence the number of dependencies between slices. A lower run-time automatically results in a higher buffer requirement for this approach. Compared to the NBSPr with core rotation, high buffer size requirements can be observed for the NBSP approach. This confirms our assumption that a constant core assignment to frame regions results in a very unbalanced core usage. The core working on the frame region with the lowest complexity leaves the other cores far behind and requires a theoretical unlimited buffer size.

For the three encoder-independent approaches, the SR approach shows the best buffer size requirements for low core counts. The horizontal partitioning and the line-wise processing from top to down results in small buffer requirements. The core count determines the number of equally processed lines and therefore also the required buffer sizes. Also the DG approach shows a low buffer size requirement which raises only slowly for higher core counts. For 8 cores, the DG approach even outperforms the SR approach.

The buffer requirements for the MC approach are between the SR and the NBSPr approaches for 2 cores. Higher core counts raise the buffer requirements since each additional core results in a smaller column width and a faster downwards moving processing order. This increases the number of simultaneously processed macroblock lines which have to be buffered. For the MC approach, sequence-dependent buffer size variations can be observed.

6. CONCLUSIONS

In this study, we have evaluated 6 data-parallel approaches for the H.264 decoding process. We compare the approaches against each other and investigate the impact of raising core counts on the decoder's run-time, core usage, transfer rates and buffer requirements.

The run-time of each parallelization approach is strongly influenced by the frame partitions' sizes and shapes. Large and dependency-minimizing partitions cause less inter-communication between the cores and a lower run-time. Additionally, a static assignment of each core to constant frame regions makes an approach highly sensitive to local complexity peaks and can result in an unequal workload balancing. Assigning each core to multiple and spatially separated frame regions (e.g. the SR approach) or rotating the assignment order between frames can compensate these complexity variations more efficiently. A rotating assignment has the advantage that it can reduce dependencies by maintaining large partitions and process diverse frame regions on each core.

For approaches working on larger partitions it is generally easier to exploit locality. This reduces transfer rates since more spatially close neighbouring macroblocks are processed on the same core and efficient data reuse is possible. Reusing the data from horizontal neighbouring macroblocks is easily possible by keeping a single macroblock's prediction and deblocking information in the local cache. Vertical caching requires that the partitions' width is smaller than the number of locally storable macroblocks. For approaches based on horizontal splitting, this results in the highest requirements on local memories.

We can observe, that the buffer size requirements of a parallelization approach are mainly influenced by the number of vertical partitions and the height of the partition blocks in the frame. A higher number of vertical splits or larger partition heights increase the buffer size requirements. Dependencies between partitions result in a higher inter-communication but also avoid cores with low complexity tasks from diverging too far from the other cores. This effect can be seen for the non-blocking SP approach where a diverging and a strong increase of the buffer size requirements can only be compensated by rotating the cores' assignment.

In future work, we want to extend our studies to higher resolutions and investigate the efficiency of parallel coding approaches for different H.264 coding tools and GOP structures. More advanced core rotation methods with an insensitivity to local complexity peaks shall be developed.

7. ACKNOWLEDGMENTS

This work has been supported by ON DEMAND Microelectronics AG [8] and the Austrian Federal Ministry of Transport, Innovation, and Technology under the FIT-IT project VENDOR (Project nr. 812429). Michael Bleyer would like to acknowledge the Austrian Science Fund (FWF) for financial support under project P19797.

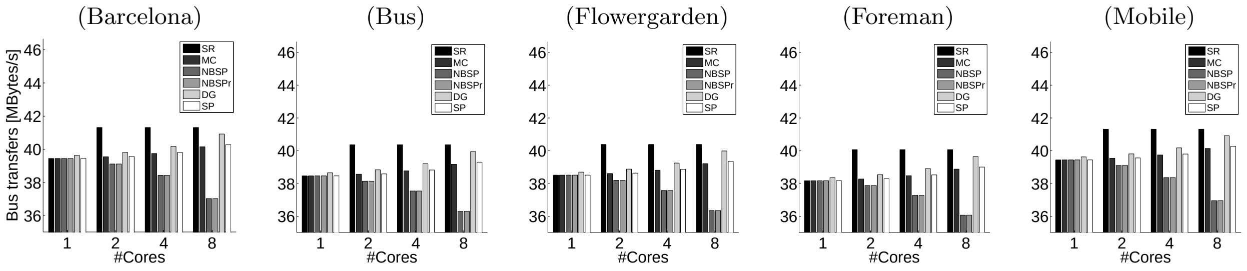


Figure 18: Data transfer volume for reference data and deblocking information, if one macroblock line can be cached in local core memory. It contains the amount of data which is transferred between the cores and the system's shared memory for the deblocking, intra-prediction and inter-prediction.

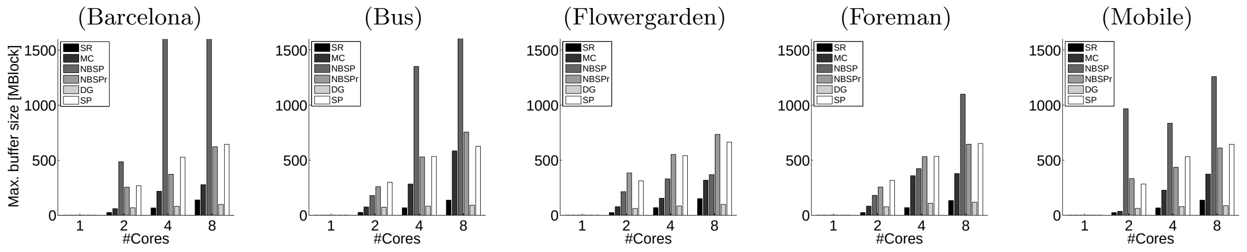


Figure 19: Required size for parser output buffer. This buffer is located between the parser and the reconstruction cores and avoids write stalls at the parser side.

8. REFERENCES

[1] E. Bondarev, M. R. Chaudron, and P. H. d. With. Predictable component-based software design of real-time MPEG-4 video applications. In *Proc. of the SPIE, Visual Communications and Image Processing*, volume 5960, pages 2288–2298, July 2005.

[2] T.-W. Chen, Y.-W. Huang, T.-C. Chen, Y.-H. Chen, C.-Y. Tsai, and L.-G. Chen. Architecture design of H.264/AVC decoder with hybrid task pipelining for high definition videos. In *Proc. of the IEEE Int. Symp. on Circuits and Systems*, pages 2931–2934, 2005.

[3] 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 *Proc. of the 18th Int. Parallel and Distributed Processing Symp.*, volume 1, pages 63–72, 2004.

[4] ITU-T and ISO/IEC. *Advanced video coding for generic audiovisual services (ITU Rec. H.264 | ISO/IEC 14496-10)*. ITU-T and ISO/IEC, March 2005.

[5] Joint Model Version 12.2. *http://iphome.hhi.de/suehring/tml/*. 07.01.2008.

[6] C. Meenderinck, A. Azevedo, M. Alvarez, B. Juurlink, and A. Ramirez. Parallel scalability of H.264. In *Proc. of the 1st Workshop on Programmability Issues for Multi-Core Computers*, January 2008.

[7] T. Moriyoshi and S. Miura. Real-time H.264 encoder with deblocking filter parallelization. In *IEEE Int. Conf. on Consumer Electronics*, pages 63–64, 2008.

[8] ON DEMAND Microelectronics AG. *http://www.odmsemi.com*. 07.01.2008.

[9] A. Rodríguez, A. González, and M. Malumbres. Hierarchical parallelization of an H.264/AVC video encoder. In *Proc. of the Int. Symp. on Parallel Computing in Electrical Engineering*, pages 363–368, 2006.

[10] K. Schöffmann, M. Fauster, O. Lampl, and L. Böszörményi. An evaluation of parallelization concepts for baseline-profile compliant H.264/AVC decoders. In *Euro-Par 2007, Parallel Processing*, pages 782–791, 2007.

[11] F. H. Seitner, R. M. Schreier, M. Bleyer, and M. Gelautz. A macroblock-level analysis on the dynamic behaviour of an H.264 decoder. In *IEEE Int. Symp. on Consumer Electronics 2007*, pages 1–5, Irving, June 2007.

[12] F. H. Seitner, R. M. Schreier, M. Bleyer, and M. Gelautz. A high-level simulator for the H.264/AVC decoding process in multi-core systems. In *Proc. of the SPIE, Multimedia on Mobile Devices 2008*, volume 6821, pages 5–12, January 2008.

[13] S. Sun, D. Wang, and S. Chen. A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition. In *Proc. of the 3rd Int. Conf. on High Performance Computing and Communications*, pages 577–585, September 2007.

[14] E. B. van der Tol, E. G. Jaspers, and R. H. Gelderblom. Mapping of H.264 decoding on a multiprocessor architecture. In *Proc. of the SPIE*, volume 5022, pages 707–718, May 2003.

[15] S. H. Wang, W. H. Peng, Y. He, G. Y. Lin, C. Y. Lin, S. C. Chang, C. N. Wang, and P. Chiang. A platform-based MPEG-4 advanced video coding (AVC) decoder with block level pipelining. In *Proc. of the 2003 Joint Conf. of the 4th Int. Conf. on Information, Communications and Signal Processing and the 4th Pacific Rim Conf. on Multimedia*, volume 1, pages 51– 55, Dec 2003.

[16] Z. Zhao and P. Liang. A highly efficient parallel algorithm for H.264 video encoder. In *Proc. of the 31st IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, volume 5, pages 489–492, May 2006.