A Multi-Threaded Full-feature HEVC Encoder Based on Wavefront Parallel Processing

Stefan Radicke[1,2], Jens-Uwe Hahn[1], Christos Grecos[2] and Qi Wang[2]
[1]*Hochschule der Medien, Nobelstraße 10, Stuttgart, Germany*
[2]*School of Computing, University of the West of Scotland, High Street, Paisley, U.K.*

Keywords: High Efficiency Video Coding (HEVC), Wavefront Parallel Processing (WPP), Multi-threading.

Abstract: The High Efficiency Video Coding (HEVC) standard was finalized in early 2013. It provides a far better coding efficiency than any preceding standard but it also bears a significantly higher complexity. In order to cope with the high processing demands, the standard includes several parallelization schemes, that make multi-core encoding and decoding possible. However, the effective realization of these methods is up to the respective codec developers.
We propose a multi-threaded encoder implementation, based on HEVC's reference test model HM11, that makes full use of the Wavefront Parallel Processing (WPP) mechanism and runs on regular consumer hardware. Furthermore, our software produces identical output bitstreams as HM11 and supports all of its features that are allowable in combination with WPP. Experimental results show that our prototype is up to 5.5 times faster than HM11 running on a machine with 6 physical processing cores.

1 INTRODUCTION

The Joint Collaborative Team on Video Coding (JCT-VC), which is a cooperation partnership of the ITU-T Video Coding Experts Group (VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG), has recently developed the High Efficiency Video Coding (HEVC) standard. Because of its excellent compression performance, it is perfectly suited for beyond-HD video resolutions like 4K or 8K Ultra High Definition (UHD) (Sullivan et al., 2012). However, the complexity of HEVC is significantly high and a traditional single-threaded encoder can therefore not provide real-time performance. For this reason, the standard describes several high-level parallelization mechanisms which allow codec developers to leverage the potential of multi-core platforms more easily. In this context, slices, tiles, and Wavefront Parallel Processing (WPP) are the three key concepts.

Video coding standards are usually defined from a hypothetical decoder's point of view and HEVC is no exception in this regard. Therefore, it is relatively straightforward to utilize the described parallel processing techniques in a software decoder. In (Chi et al., 2012), for example, the authors provide a comprehensive analysis of tiles and WPP and propose a decoder implementation with real-time performance.

The encoder side, on the other hand, imposes additional challenges for parallel hardware- and software-architectures. For example, intra- and inter-mode decision, as well as numerous cost functions, are complex modules that need to be made thread-safe. They also must be specifically optimized and fine-tuned for parallel execution. Additionally, encoding is in general more computationally expensive than decoding and it is hence more difficult to achieve real-time speed.

Yan *et al* present a highly-parallel implementation of Motion Estimation (ME) based on Motion Estimation Regions (MERs) running on a specific 64-core processor (Yan et al., 2013). In principle, MERs allow the calculation of ME for multiple Prediction Units (PUs) within a single Coding Tree Unit (CTU) concurrently. Yan *et al* extend the approach with the utilization of a directed acyclic graph which transparently models the dependencies between neighboring CTUs in order to enable a significantly higher degree of parallelism than with MERs alone. The result is an efficient and very fine-grained parallel algorithm, which is particularly well suited for many-core processors. Such devices are, however, not widely used in the consumer market. Furthermore, the work only addresses ME and not the entire encoding loop. In summary, for software encoders used by regu-

Radicke S., Hahn J., Grecos C. and Wang Q..
A Multi-Threaded Full-feature HEVC Encoder Based on Wavefront Parallel Processing .
DOI: 10.5220/0005018300900098
In *Proceedings of the 11th International Conference on Signal Processing and Multimedia Applications* (SIGMAP-2014), pages 90-98
ISBN: 978-989-758-046-8

lar end-users, a more coarse-grained parallel algorithm, which covers the entire encoding loop, would be preferable.

In contrast, the work proposed by Zhao *et al* is more specifically targeted towards consumer-level computers (Zhao et al., 2013). They present a parallel intra-prediction algorithm based on an open source implementation of HEVC. The proposed intra-prediction method is loosely based on the WPP scheme, while the entropy coding is done in raster scan order by a separate thread. This leads to high speedups with minimal coding loss. Unfortunately, a considerable number of features are missing in their prototype compared to the HEVC reference software. Most noticeably, inter-prediction is not supported, nor some other tools such as Sample Adaptive Offset (SAO) or non-square PUs, for example.

The described scenario motivated us to implement a full-feature encoder based on the HEVC reference software HM11 (Bossen et al., 2013) that runs on regular multi-core hardware. Our proposed prototype is is based on the WPP scheme, which makes it possible for our software to scale naturally with the number of available processing cores and with increased video resolution. On our 6-core test machine, we achieve speedups of up to 5.5x compared to HM11, with identical video quality and compression performance. Furthermore, all encoder features, tools, and algorithms are fully supported, with the exception of those, that are not allowed to be used in combination with WPP.

The rest of the paper is structured as follows: Section 2 outlines the high-level parallelization schemes of HEVC, Section 3 provides a theoretical analysis of the speedups achievable with WPP, and Section 4 describes the implementation of our encoder prototype. Experimental results can be found in Section 5 and Section 6 concludes the work.

2 HIGH-LEVEL PARALLELISM IN HEVC

Every video encoder exploits redundancies and statistical characteristics in image sequences in order to compress the data as much as possible. However, this introduces various dependencies within the bitstream, between neighboring frames, and between individual coding blocks. The blocks of a video sequence must hence be processed in specific sequential order. Typically, raster scan order is used. In parallel processing environments, some of these dependencies must be broken in order to enable concurrency, which unavoidably reduces the compression performance.

HEVC provides several low- and high-level mechanisms for dependency removal (Choi and Jang, 2012), whereby only the latter are of interest for this work. Namely these are slices, tiles and WPP, which all subdivide the video frames in certain ways. In general, every picture is subdivided into CTUs which have a maximum size of 64x64 luma pixels. They can be recursively split into square-shaped Coding Units (CUs), which in turn contain one or more PUs and Transform Units (TUs) (Kim et al., 2013). It is important to note that all the high-level parallelization schemes operate at the CTU level.

2.1 Slices

Each picture comprises one or multiple slices, which can be decoded independently from one another. The slices themselves consist of a sequence of CTUs and may hence not be rectangular in shape. Slices may have individual headers within the bitstream or, in the case of dependent slice segments, infer their respective syntax elements from previous slices.

The main purpose of slices is to deal with network and transmission problems like packet loss, for example. Even though slices can be utilized for parallel processing, they are usually not the preferable solution. One reason for this is that it can be difficult to find a slice configuration that is suitable for both network packetization and parallel computing at the same time. Another issue is the relatively high loss in Rate-Distortion (RD) performance caused by the fact that slices break spatial as well as statistical dependencies. In case of independent slices, for instance, it is not possible to perform prediction across slice boundaries and the entropy contexts are reinitialized at the beginning of every slice.

2.2 Tiles

In contrast to slices, tiles were specifically designed as a tool for parallelization. A picture gets logically subdivided into rectangular regions containing an integer number of CTUs which can be coded independently from one another. To make this possible, prediction dependencies are broken at tile boundaries. Within each tile, the CTUs are processed in raster scan order and the entropy coding state is reset at the start of every tile. Slices and tiles may also be combined as long as certain restrictions are met (Bossen et al., 2013).

Tiles make it possible to leverage different multi-core architectures effectively, as the picture can be structured in a way that fits the respective platform best. The size and position of the tiles may be chosen with respect to the available processing cores and

their individual computational capabilities. Specific characteristics of the video footage can also be taken into account. Areas with a rich amount of detail, which typically take longer to encode, can be processed using smaller tiles in order to achieve good load balancing, for example. This high degree of flexibility, however, can also be of disadvantage, because special effort must be made to find an optimal partitioning for every hardware platform and video sequence. Another issue with tiles is that they can introduce visible artifacts at their boundaries caused by the broken prediction dependencies. Further information about tiles can be found in (Misra et al., 2013).

2.3 Wavefront Parallel Processing

The WPP principle maintains all dependencies with the exception of the statistical ones and is hence by far the best method in terms of quality and coding loss. This is made possible by processing all CTUs in a natural order so that their interdependencies are never broken. Figure 1 exemplarily shows how a picture is processed using four threads. It also illustrates that every CTU depends on its left, top-left, top, and top-right neighbor, which is indicated by the grey arrows decorating the rightmost CTU of the second row. Another important feature of WPP is that the entropy coder state for every line is inherited with an offset of two CTUs, as represented by the diagonally downwards pointing grey arrows at the left side of the figure. For example, the Context Adaptive Binary Arithmetic Coding (CABAC) state is stored after the first two CTUs of line one have been fully coded and is then used as the initial entropy coder state for line two. In summary, this means that the individual lines of a picture can be processed independently, as long as an offset of at least two CTUs is kept for every consecutive line.

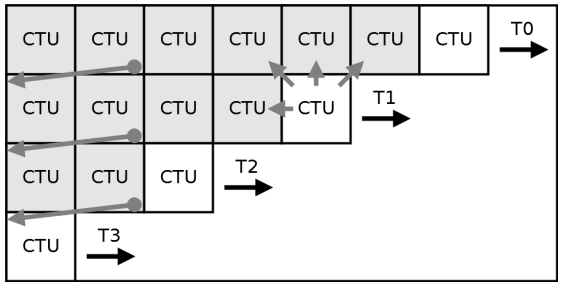


Figure 1: Principle of WPP and CTU Dependencies.

In addition to its negligible coding losses, WPP has several other benefits compared to slices and tiles. The main advantage is that the scheme scales naturally with the video resolution and with the number of available processing units. In the first case, more independent CTU lines exist, and in the latter case, more of the available lines can be coded simultaneously. Furthermore, the locality of the memory accesses is largely unaffected, since consecutive CTUs are processed in raster scan order as usual, which results in good cache efficiency. It should also be mentioned that a WPP-encoded bitstream can easily be decompressed by a single-threaded decoder. All that is needed is some additional memory to store the respective CABAC state after the first two CTUs of the current line, which is then used as the initial state for the following line.

The positive characteristics of WPP motivated us to build our prototype based on it. However, tiles are not supported by our software, because HEVC prohibits the combination of WPP and tiles for the sake of design simplicity (Sullivan et al., 2012). It should be noted, though, that our extended HM11 test model can still be used as a flexible basis for algorithm design and evaluation, since all its other features are unaffected by our implementation.

3 THEORETICAL ANALYSIS OF WPP

Even though WPP is very good in terms of scaling, the actual amount of parallelism, and hence also the possible speedups, are not quite obvious. In comparison, if four equally sized tiles per picture are used and four Central Processing Units (CPUs) are available, the maximum speedup of the CTU-loop is 4x. That is, assuming every CTU takes roughly the same time to process. With WPP, however, the picture partitioning directly depends on the video resolution and there is no direct correlation between the number of cores and the maximum concurrency. If four CPUs are used, as in the above example, the CTU-loop will not automatically be up to 4x faster with WPP. This is because the prediction-dependencies are being maintained, which implies that each CTU can only be processed after its respective left, top-left, top, and top-right neighbors. Therefore, not all of the four available cores will be busy at any given point in time since they often have to wait for one another. Simply put, a lower CTU-line can never overtake a higher one, which limits the amount of concurrency due to the requirement of frequent synchronization. If, on the other hand, more CPUs are available, the speedup will automatically scale in respect of the available video resolution. This would, in turn, not be the case with tiles. For these reasons, we provide a detailed analysis of what speedups can theoretically be achieved with WPP in the upcoming paragraphs.

The first thing that needs to be addressed is the ideal number of threads. It is directly related to the respective video resolution and can be calculated using the following equation:

$$T_{ideal} = min\left(ceil\left(\frac{X_{CTUs}}{2}\right), Y_{CTUs}\right) \quad (1)$$

The function $min()$ selects the lower one of the two input values and the function $ceil()$ rounds the given value up to the nearest integer. The parameters X_{CTUs} and Y_{CTUs} represent the width and the height of the picture measured in CTUs. The value of X_{CTUs} is divided by 2 because of the offset that needs to be maintained for every consecutive row. Therefore, usually the width of the video is the limiting factor for parallelism, except in the case of ultra widescreen formats where $width \geq 2 \times height$. It should further be noted that the validity of this equation does not depend on the number of pixels in the horizontal and vertical dimensions of the CTU. For our experiments we used a CTU-size of 64x64 luma samples and hence get $T_{ideal} = 15$ for 1920x1080p, and $T_{ideal} = 20$ for 2560x1600p videos, respectively.

1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10
2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	10	10
3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	10	10	10	10
4	4	5	5	6	6	7	7	8	8	9	9	10	10	10	10	10	10	9	9
5	5	6	6	7	7	8	8	9	9	10	10	10	10	10	10	9	9	8	8
6	6	7	7	8	8	9	9	10	10	10	10	10	10	9	9	8	8	7	7
7	7	8	8	9	9	10	10	10	10	10	10	9	9	8	8	7	7	6	6
8	8	9	9	10	10	10	10	10	10	9	9	8	8	7	7	6	6	5	5
9	9	10	10	10	10	10	10	9	9	8	8	7	7	6	6	5	5	4	4
10	10	10	10	10	10	9	9	8	8	7	7	6	6	5	5	4	4	3	3
10	10	10	10	9	9	8	8	7	7	6	6	5	5	4	4	3	3	2	2
10	10	9	9	8	8	7	7	6	6	5	5	4	4	3	3	2	2	1	1

Figure 2: Amount of Parallelism at Every CTU-Position.

The next thing to consider is the actual amount of parallelism that can be achieved, or in other words, how many CTUs are being processed concurrently at any given point in time. Figure 2 visualizes the degree of concurrency at every CTU-position using a heatmap. For simplicity, a relatively small resolution of 1280x720p is assumed. The numbers indicate how many CTUs can theoretically be processed in parallel at any position within the frame. This directly translates to the number of potentially active threads at the respective positions. It can be seen that the parallelism is low at the beginning of the frame, reaches its maximum in the middle, and then diminishes towards the end of the frame. Unfortunately, the time window in which T_{ideal} threads are active, which is 10 in this case, is rather small.

With this information we can calculate the theoretical speedup of the CTU-loop S_{max}, by dividing the sum of the processing times of all CTUs by the time it would take to process the entire frame in a single thread:

$$S_{max} = \frac{\sum_{x,y} t(CTU_{x,y})}{t_{CTU} \times X_{CTUs} \times Y_{CTUs}} \quad (2)$$

The symbol t_{CTU} denotes the time it takes to compute one CTU in a single-threaded fashion. We assume here that every CTU takes equally long to process. On real hardware, however, this will most definitely not be the case, since the performance is typically limited by shared caches, memory transfers, scheduling overhead, and other factors. However, as we are trying to find the maximum theoretical speedup, the assumption of an ideal parallel computer with no execution overhead is sufficient. The function $t(CTU_{x,y})$ calculates the multi-threaded processing time of a given CTU based on the amount of parallelism at the respective position x,y. For example, 2 CTUs can be computed concurrently at position $x = 0, y = 2$. Therefore, the call $t(CTU_{0,2})$ returns $\frac{t_{CTU}}{2}$, because it is assumed that it takes just as long to process 2 CTUs in parallel, as it would take to process a single one alone.

For 2560x1600p sequences using 6, 12, and $T_{ideal} = 20$ threads, S_{max} results in 5.00x, 8.33x, and 11.36x, respectively. It can be concluded, that the possible gains with WPP are significant. It also has to be emphasized that these are theoretical speedups of the CTU-loop and not of the entire system. Not all parts of the encoder can be parallelized with WPP. This is due to the fact that aspects such as the reference list construction, some image filters, and the concatenation of the various substreams still need to be processed sequentially.

4 IMPLEMENTATION

The HM11 encoder software can be configured to produce a bitstream that can be decoded in a wavefront like fashion using multiple computational cores by setting the option *WaveFrontSynchro = true* (Kim et al., 2013). In this mode, the CABAC state present after the first two CTUs of each line is temporarily stored and then used as the initial state of the respective next line. The software also maintains multiple bitstreams, one per CTU-row, which are concatenated at the end of each picture to generate the final output data. However, all this is done in a single thread without any parallelism.

The implementation we propose is based on the HM11 software and extends it to a real parallel encoding framework, which scales naturally with the available CPUs and with higher video resolutions.

4.1 Bitstream Verification

It is well known that parallel computing can be very error-prone because of issues like data races or deadlocks. In order to guarantee that our prototype does not contain any threading-related problems, we first implemented a component called *BitstreamVerifier*. It allows us to compare the generated output data to a reference bitstream that was created in advance with the unmodified HM11 software during runtime. With this measurement, we were able to detect errors as soon as they arose during development and could thus keep the turnaround times reasonably short. The *BitstreamVerifier* was of course disabled for the final experiment runs, since it unavoidably causes some computational overhead.

4.2 Thread Pool

In order to make our prototype scalable and flexible, we implemented a thread pool system that can manage an arbitrary amount of worker threads. Typically, the degree of parallelism is set according to the number of available processing cores, but any other number of threads can also be used. The individual CTU-lines are kept in an ordered task-queue which is protected against race conditions using mutual exclusion. The worker threads autonomously grab the CTU-lines out of the list and process all their CTUs accordingly. In case there are more CTU-lines than threads, which is typical, each worker might process multiple lines. The thread pool management system itself waits until all threads have finished their work, which means the entire CTU-loop is finished.

4.3 Entropy Contexts

To make parallel encoding possible, each line needs its own entropy coder context to work with. Consequently, instead of using one temporary state to realize the probability synchronization, as described at the very beginning this section, the individual contexts must be used. This means, the state after the second CTU of line *n* is directly loaded into the CABAC state of line *n+1*. After that, line *n+1* can be processed by its associated worker thread. Fortunately, the HM11 software is already prepared to work with multiple entropy coder contexts, thus we could modify the source code to fit our needs, accordingly.

4.4 Synchronization

When the rows are actually processed in parallel, the offset of 2 CTUs, relative to the respective next row, must be maintained all the time. This allows the intra- and inter-prediction routines to fully exploit all spatial correlations because the required neighboring blocks are guaranteed to be available. We utilize semaphores to implement the synchronization implicitly. Every line has its own semaphore, which is calculated after every finished CTU in the following way:

$$S_C = \begin{cases} X_{CTUs}, & \text{if } P_{CTUs} = X_{CTUs} \\ S_C + 1, & \text{if } P_{CTUs} \geq 2 \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

Here, S_C refers to the semaphore's counter, X_{CTUs} is the width of the picture measured in CTUs, and P_{CTUs} is the number of CTUs that have already been processed in the given row. The respective subsequent row synchronizes with the semaphore and decrements its counter by 1, if possible. This ensures that the mandatory offset of 2 CTUs is kept all the time. The only exception is the first case in the above equation, which happens when the line is completely finished and the offset is no longer required. For example, with a CTU-line length of 6, each semaphore would count [0, 0, 1, 2, 3, 4, 6]. During processing, however, the counters are decremented by the respective blocked threads as soon as possible, which is why their values typically toggle between 0 and 1.

4.5 Thread-safe Data Structures and Algorithms

The source code of the HEVC test model was not designed to be executed concurrently, which means that its classes and functions are not thread-safe by default. The easiest way to make existing single-threaded code executable in parallel is to add various synchronization points, which ensure a valid order of execution and data accesses and thereby eliminate the possibility of race conditions. However, it is well known that mutual exclusion can easily become a severe bottleneck in any parallel program. We therefore decided to use lock-free programming techniques to ensure thread-safety and to achieve optimal performance at the same time.

In order to avoid synchronization completely, we duplicate all needed encoder modules for every CTU-line, so that they can work totally independent from one another. It is generally known that data races can only occur if multiple threads operate on the

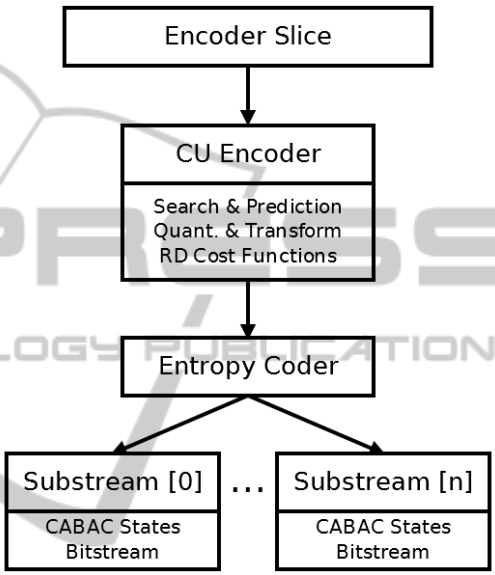
same data. If several threads write into the same segment of memory concurrently, the result is non-deterministic. Take, for example, HEVC's Motion Vector (MV) refinement algorithms. An Interpolation Filter (IF) is utilized to generate the needed subpixels for the respective fractional MVs. The resultant subpixel-blocks are stored in a number of temporary buffers and serve as a basis for the cost estimation. In a multi-threaded environment, these buffers must be duplicated, so that every thread can work on its own individual area of memory independently. Otherwise, the threads would randomly overwrite each other's data, which would ultimately lead to race conditions and therefore false results.

The *entropy coder* and the *CU encoder*, which itself contains modules for *search*, *prediction*, *quantization*, *transform* and *RD cost computation*, are the subsystems that need to be copied. To do this, we employ a deep-copy mechanism with subsequent cross-reference reconstruction. Deep-copy essentially means that not only the instance of a class is cloned, but all its respective member objects as well. Accordingly, new memory needs to be allocated for every class member and their data must be copied respectively. The results are completely independent new objects with no shared members, buffers, or pointers. This circumstance makes it obvious why the mentioned cross-reference reconstruction is necessary. Some of the new objects need to associate each other so that they can properly work together. The *CU encoder* needs to know its *search* module, for example. Therefore, all needed aggregations and references between the newly cloned objects need to be set correctly.

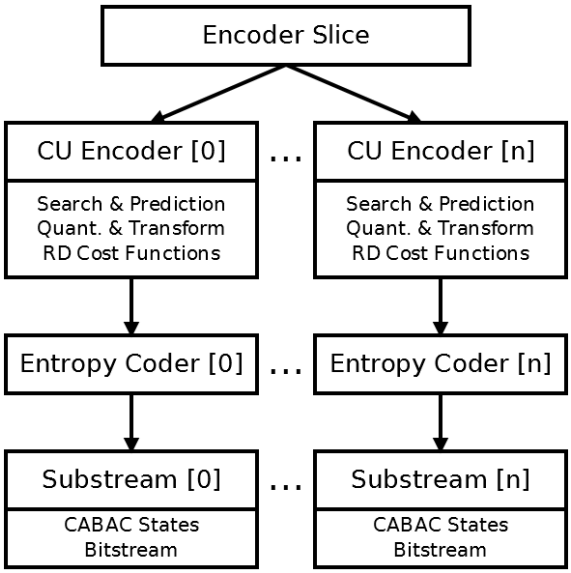
The following steps are performed to deep-copy the individual components and their sub-modules before the worker threads can start:

1. Create byte-identical copies of the class instances.
2. Re-allocate memory for all the temporary member data structures and sub-modules they have.
3. Copy the contents of the respective sub-modules.
4. Eventually invoke the objects' initialization functions with proper parameters.
5. Set all cross-references between the new objects.
6. Associate the objects with their respective entropy coders, CABAC states, and bitstreams.

Figure 3 shows the differences between a single-threaded encoder (a) and our parallel framework (b). As explained earlier, the subsystems and their respective memory buffers are duplicated for every thread. Each thread thus has its own dedicated class instances and memory areas to operate on. Therefore, the threads can work independently from one another. They only need to be synchronized in order to stay within the limitations of the WPP scheme as described in Section 4.4. It is also worth noting that the WPP substreams do not need to be duplicated as the original HM11 software already has multiple instances of them.



(a) Regular Encoder.



(b) Parallel Encoder.

Figure 3: Comparison of a Regular Single-Threaded HM11 Encoder and the Proposed Parallel Architecture.

5 EXPERIMENTAL RESULTS

We used the following development environment for the experiment runs: Intel Core i7-3930K CPU@3.20 GHz, 32 GB of Random Access Memory (RAM), and Windows 7 Professional 64-Bit operating system. The Core i7-3930K CPU has 6 physical cores and features Hyper-Threading (HT) technology, which means it has two Architectural States (AS) per core. This makes 12 logical cores available to the operating system and also increases the processor's execution resource utilization due to faster context switches.

On this platform, we encoded four HEVC test sequences: *Kimono1* (1), *ParkScene* (2), *PeopleOnStreet* (3), and *Traffic* (4). (1) and (2) have a resolution of 1920x1080p and (3) and (4) of 2560x1600p. All videos use a color sub-sampling of 4:2:0. According to the common test conditions (Bossen, 2013), we used the profiles *random access* (ra), *intra* (in), and *low delay* (ld), with Quantization Parameters (QPs) 22, 27, 33, and 37. The number of worker threads was set to 6, 8, 10, 12, and T_{ideal}, as identified in Section 3, was 15 for 1080p and 20 for 1600p videos. Using more threads is unreasonable, because the amount of parallelism would remain the same. For all multi-threaded test runs we set *WaveFrontSynchro = true* and the output of the unmodified HM11 encoder serves as basis for comparison.

Table 1: RD Performance of WPP.

Cfg.	Seq.	BD-PSNR Y, U, V [dB]	BD-Rate Y, U, V [%]
(ra)	(1)	-0,037, -0,022, -0,021	1.2, 1.4, 1.1
	(2)	-0.021, -0.016, -0.022	0.6, 0.8, 1.1
	(3)	-0.024, -0.024, -0.015	0.5, 1.0, 0.7
	(4)	-0.023, -0.014, -0.015	0.7, 0.7, 0.8
(in)	(1)	-0.011, -0.005, -0.005	0.3, 0.2, 0.2
	(2)	-0.003, -0.008, -0.007	0.1, 0.3, 0.3
	(3)	-0.007, -0.003, -0.001	0.1, 0.1, 0.0
	(4)	-0.002, -0.006, -0.008	0.0, 0.2, 0.2
(ld)	(1)	-0.036, -0.027, -0.035	1.1, 1.6, 1.9
	(2)	-0.028, -0.025, -0.012	0.9, 1.3, 0.7
	(3)	-0.023, -0.040, -0.035	0.5, 1.7, 1.8
	(4)	-0.026, -0.039, -0.025	0.8, 2.3, 1.5

Table 1 summarizes the RD performance results. We calculated the differences in Peak Signal-to-Noise Ratio (PSNR) and bitrate over the four different QPs using the Bjøntegaard Delta (BD) metric (Bjøntegaard, 2001). The BD-metric is a well-known method to determine the average PSNR and bitrate differences between two RD-plots. We computed the respective average differences between the output of our prototype and the output of HM11 with WPP disabled, using the four QPs listed above. Consequently, the results presented in Table 1 show one BD-PSNR and one BD-Rate value for every color channel, instead of listing the data for all used QPs individually. This is mainly for better clarity and readability. It is clearly obvious that the losses caused by WPP are negligible, because only the statistical dependencies are affected by it, while the temporal and spatial ones remain intact. We would like to emphasize once again that our prototype produces identical bitstreams as the HM11 encoder with the setting *WaveFrontSynchro = true*, so these results are easily reproducible.

The most important aspect of our implementation are of course the speedups *S*, which we calculated by dividing the time HM11 needed to process the respective sequence, t_{HM11}, by the time it took using our prototype, t_{WPP}.

$$S = \frac{t_{HM11}}{t_{WPP}} \tag{4}$$

The complete data set of all test runs can be found in Table 2. In addition, the average speedups over the utilized QPs are visualized as diagrams in Figure 4. We computed the average speedups S_{AVG} as

$$S_{AVG} = \frac{S_{QP22} + S_{QP27} + S_{QP32} + S_{QP37}}{4}, \tag{5}$$

where each parameter S_{QPn} represents the speedup of a single experiment run as listed in Table 2. For example, the average speedup the experiment *Cfg. (ra), Seq. (1), QPs {22, 27, 32, 37}, 6 Threads*, was computed as $S_{AVG} = \frac{4.04+3.99+4.03+3.99}{4} = 4.01$.

It is clearly noticeable that the speedups we achieved are significant among all configurations. They range roughly between 4.5x and 5.5x with the maximum number of threads used. As the analysis in Section 3 suggests, the gains are bigger for higher resolutions, because more CTU-lines are available for concurrent processing. However, the characteristics of the speedup-curves are very interesting, considering that video encoding is in general an extremely compute-bound application, with a typically very high processor utilization. It would seem that HT is not of much benefit in this context, but our results show otherwise. The performance jump from 6 to 12 threads is very high, even though our CPU only has 6 physical cores. The fast context switches due to HT technology can help here, because of the variable completion times and the significant overhead caused by memory transfers. Since two AS are available per processing core, each one of them can simultaneously manage two threads. Therefore, if two threads reside on the same core and one of them is finished early,

Table 2: Speedups of WPP.

Cfg.	Seq.	QP	HM11 Time[s]	Speedups for N Threads 6, 8, 10, 12, ideal (15 or 20)
(ra)	(1)	22	12866	4.04, 4.51, 4.62, 4.75, 4.75
		27	10950	3.99, 4.48, 4.61, 4.77, 4.80
		32	9741	4.03, 4.43, 4.62, 4.79, 4.84
		37	8921	3.99, 4.42, 4.62, 4.80, 4.84
	(2)	22	11925	3.90, 4.30, 4.43, 4.56, 4.58
		27	9977	3.92, 4.36, 4.46, 4.62, 4.61
		32	8882	3.98, 4.39, 4.48, 4.66, 4.66
		37	8255	3.95, 4.43, 4.56, 4.73, 4.72
	(3)	22	20177	4.17, 4.77, 5.02, 5.29, 5.39
		27	17001	4.17, 4.82, 5.03, 5.33, 5.42
		32	14861	4.20, 4.81, 5.02, 5.32, 5.40
		37	13393	4.22, 4.84, 5.03, 5.33, 5.40
	(4)	22	13430	4.22, 4.75, 5.01, 5.29, 5.35
		27	11291	4.25, 4.80, 5.04, 5.32, 5.43
		32	10185	4.22, 4.80, 5.02, 5.29, 5.42
		37	9608	4.18, 4.80, 5.02, 5.28, 5.43
(in)	(1)	22	3216	3.75, 4.16, 4.37, 4.53, 4.58
		27	2659	3.74, 4.10, 4.30, 4.48, 4.51
		32	2405	3.68, 4.06, 4.27, 4.47, 4.51
		37	2274	3.69, 4.11, 4.32, 4.51, 4.55
	(2)	22	3910	3.69, 4.13, 4.32, 4.48, 4.54
		27	3213	3.72, 4.12, 4.30, 4.48, 4.53
		32	2750	3.68, 4.12, 4.30, 4.49, 4.53
		37	2432	3.69, 4.09, 4.28, 4.47, 4.53
	(3)	22	4548	3.84, 4.36, 4.58, 4.89, 5.10
		27	3866	3.78, 4.33, 4.56, 4.84, 5.07
		32	3409	3.80, 4.29, 4.53, 4.83, 5.02
		37	3091	3.82, 4.29, 4.51, 4.82, 4.99
	(4)	22	4465	3.86, 4.40, 4.68, 4.99, 4.98
		27	3789	3.83, 4.39, 4.65, 4.97, 4.97
		32	3350	3.82, 4.37, 4.62, 4.96, 4.97
		37	3045	3.81, 4.28, 4.60, 4.93, 4.96
(ld)	(1)	22	19489	4.14, 4.59, 4.70, 4.80, 4.83
		27	16811	4.09, 4.59, 4.72, 4.83, 4.84
		32	14901	4.13, 4.59, 4.72, 4.86, 4.88
		37	13447	4.09, 4.59, 4.73, 4.86, 4.89
	(2)	22	18421	4.01, 4.45, 4.56, 4.69, 4.73
		27	15242	3.99, 4.41, 4.52, 4.67, 4.70
		32	13346	4.00, 4.42, 4.54, 4.70, 4.70
		37	12193	4.06, 4.45, 4.60, 4.75, 4.75
	(3)	22	28965	4.22, 4.85, 5.08, 5.34, 5.45
		27	24653	4.24, 4.88, 5.10, 5.37, 5.45
		32	21938	4.24, 4.86, 5.10, 5.38, 5.46
		37	20050	4.26, 4.86, 5.12, 5.40, 5.50
	(4)	22	19973	4.20, 4.83, 5.14, 5.39, 5.41
		27	16724	4.30, 4.84, 5.13, 5.40, 5.47
		32	14935	4.31, 4.84, 5.11, 5.37, 5.47
		37	13829	4.23, 4.84, 5.07, 5.33, 5.45

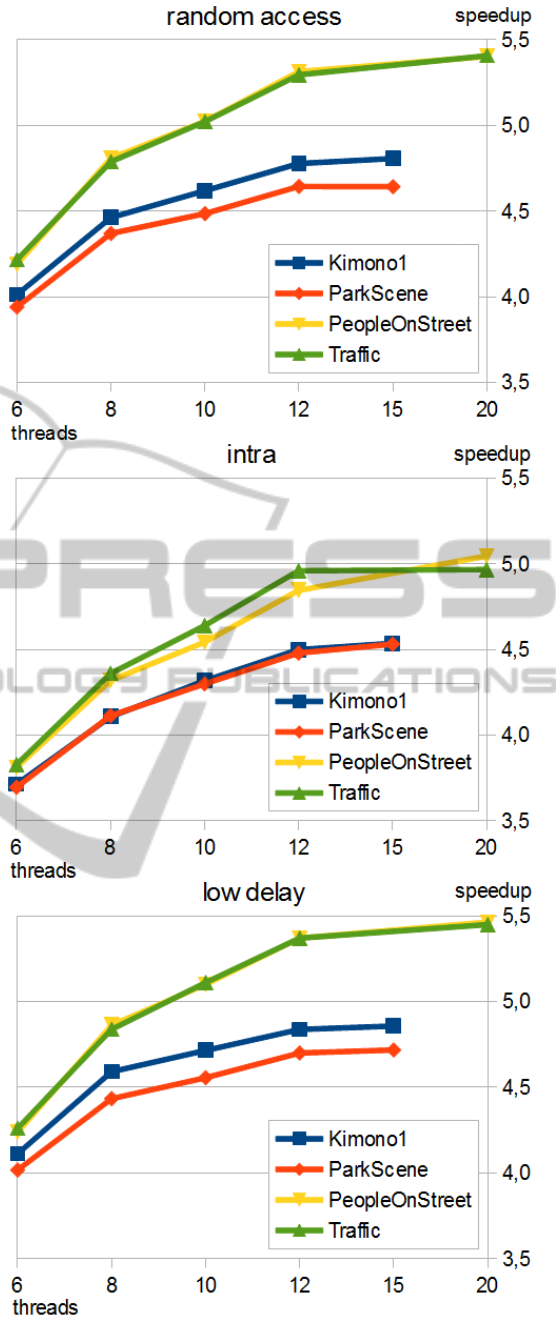


Figure 4: Average Speedups of WPP for the Profiles (random access), (intra), and (low delay).

or has to wait for a memory transfer, the other thread can immediately take over and do its work. The usually large overhead caused by context switches is thus severely reduced. The small improvements on the graphs between 12 and T_{ideal} threads suggest that the described conditions still hold true, even with slower context switches. The switches are overall slower in this case, because, more often than before, the execution contexts of some threads need to be temporarily stored in and eventually be reconstructed from the RAM. Finally, it can be seen that intra configurations benefit less from WPP overall. This is due to the fact that if only intra-prediction is used, the CTU-loop is less complex and contributes a considerably smaller proportion to the total processing time. Thus smaller overall time savings are achieved.

6 CONCLUSION

In this paper, we present a multi-threaded full-feature High Efficiency Video Coding (HEVC) encoder based on Wavefront Parallel Processing (WPP), which runs on regular consumer hardware. In addition, we provide a detailed theoretical analysis of the WPP scheme, showing its potential for significant speedups, and outline some implementation details of our parallel encoder framework. Experimental results show that our software gives speedups of up to 5.5 times on a 6-core CPU. It is noted that the proposed framework is fully compliant with the standard reference test model, as it produces identical output bitstreams and maintains the same full encoding features.

REFERENCES

Bjøntegaard, G. (2001). Calculation of Average PSNR Differences Between RD-Curves (VCEG-M33). VCEG Meeting (ITU-T SG16 Q.6), Apr. 2001.

Bossen, F. (2013). Common Test Conditions and Software Reference Configurations. Document: JCTVC-L1100, Jan. 2013.

Bossen, F., Flynn, D., and Sühring, K. (2013). HM Software Manual. Document: JCTVC-M1010, May 2013.

Chi, C., Alvarez-Mesa, M., Juurlink, B., Clare, G., Henry, F., Pateux, S., and Schierl, T. (2012). Parallel Scalability and Efficiency of HEVC Parallelization Approaches. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1827–1838.

Choi, K. and Jang, E. (2012). Leveraging Parallel Computing in Modern Video Coding Standards. *IEEE MultiMedia*, 19(3):7–11.

Kim, I., McCann, K., Sugimoto, K., Bross, B., and Han, W. (2013). High Efficiency Video Coding (HEVC) Test Model 11 (HM11) Encoder Description. Document: JCTVC-M1002, Apr. 2013.

Misra, K., Segall, A., Horowitz, M., Xu, S., Fuldseth, A., and Zhou, M. (2013). An Overview of Tiles in HEVC. *IEEE Journal of Selected Topics in Signal Processing*, 7(6):969–977.

Sullivan, G., Ohm, J., Han, W.-J., and Wiegand, T. (2012). Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668.

Yan, C., Zhang, Y., Dai, F., and Li, L. (2013). Highly Parallel Framework for HEVC Motion Estimation on Many-Core Platform. In *Proc. Data Compression Conference (DCC)*, pages 63–72.

Zhao, Y., Song, L., Wang, X., Chen, M., and Wang, J. (2013). Efficient Realization of Parallel HEVC Intra Encoding. In *Proc. International Conference on Multimedia and Expo Workshops (ICMEW)*, pages 1–6.