Accepted in the International Conference on Consumer Electronics, Communications and Networks (CECNet 2012), 2012

A Multi-Frame and Multi-Slice H.264 Parallel Video Encoding Approach with Simultaneous Encoding of Prediction Frames

Jean-François Franche, Stéphane Coulombe
Department of Software and IT Engineering, École de technologie supérieure, Université du Québec
Montréal, Canada
Jean-Francois.Franche.1@ens.etsmtl.ca, Stephane.Coulombe@etsmtl.ca

***Abstract*—This paper describes a novel multi-frame and multi-slice parallel video encoding approach with simultaneous encoding of predicted frames. The approach, when applied to H.264 encoding, leads to speedups comparable to those obtained by state-of-the-art approaches, but without the disadvantage of requiring bidirectional frames. The new approach uses a number of slices equal or greater than the number of cores used and supports three motion estimation modes. Their combination leads to various tradeoffs between speedup and visual quality loss. For an H.264 baseline profile encoder based on Intel IPP code samples running on a two quad core Xeon system (8 cores in total), our experiments show an average speedup of 7.20x, with an average quality loss of 0.22 dB (compared to a non-parallelized version) for the most efficiency motion estimation mode, and an average speedup of 7.95x, with a quality loss of 1.85 dB for the faster motion estimation mode**

Keywords- H.264; parallel algorithms; multi-core processor

I. INTRODUCTION

H.264 is the most recent and efficient standard for video compression [1]. The new tools introduced by this standard improve on the compression ratio of its predecessors by a factor of two, but at the cost of higher complexity. H.264 is also one of most commonly used formats in processing high definition (HD) videos. However, this standard is so complex that some video encoders face difficulties encoding HD video sequences in real-time, even on recent processors.

To reduce the encoding time, most H.264 video encoders developed for general processors exploit parallel approaches. For example, Intel's IPP H.264 video encoder is based on a multi-slice parallel approach, where the slices of a frame are encoded in parallel [2]. The multicore era does offer the promise of a great future for well-designed parallel approaches. A parallel approach must achieve speedups close to the number of cores used, support a variable number of cores with linear scalability, reduce processing latency, preserve or slightly reduce visual quality video, and should not change or force encoding parameters.

In this paper, we describe a novel multi-frame and multi-slice parallel approach for an H.264 video encoder. This approach includes three motion estimation (ME) modes and a support for a number of slices equal or greater than the number of cores used. The configuration of the modes and the number of slices offer various tradeoffs between speedup and visual quality loss. Experiments on 720p sequences show that the proposed approach provides speedups comparable to those obtained by state-of-the-art approaches, but without the disadvantage of requiring bidirectional (B) frames. This is important since B frames are not supported by the H.264 baseline profile, and increase latency. This property makes our approach usable in a real-time video transmission context like video-conference. The speedups obtained by the proposed method are linear with the number of cores used, and are close to the theoretical maximal speedup (i.e. close to the number of cores used). The number of cores is configurable and this approach's best motion estimation mode preserves the video quality (similar quality as a non-parallel encoder).

This paper is organized as follows. In section II, we present state-of-the-art parallel approaches. In section III, we describe the proposed parallel approach. We present experimental results in section IV, and conclude in section V.

II. STATE OF THE ART PARALLEL APPROACHES

An H.264 encoder can be parallelized into multiple threads by using either functional decomposition or data domain decomposition [3-7]. Functional decomposition divides an encoder into multiple tasks (motion estimation, transformation, entropy coding, etc.). Tasks are grouped in balanced processing groups, and each group is associated to a thread. This type of decomposition can deliver good speedup, but rarely allows good scalability and flexibility to change [3].

Data domain decomposition exploits the H.264 hierarchical structure to parallelize an encoder. This hierarchical structure has six levels: sequence, groups of pictures (GOPs), pictures, slices, macroblocks (MBs) and blocks. Each GOP starts with an instantaneous decoder refresh (IDR) frame. Frames following an IDR frame may not refer to frames preceding the IDR frame, meaning that GOPs are independent of each other, and can be encoded in parallel. Each frame belongs to one of the following types: I frames, P frames or B frames. I frames are coded without reference to any other frame, while P frames are coded with reference to a past frame, and B frames are commonly coded with reference to a past and a future frame. However, no frame depends on a B frame. A slice represents a spatial region of a coded frame. Similarly to frames, we have three types of slices: I slices, P slices and B Slices. Slices belonging to the same frame are also independent of one another, and can therefore be encoded in parallel. Slices are composed of macroblocks, and a macroblock is coded with reference to its three upper neighbors and its left neighbor.

In [4], Changying et al. describe a parallel algorithm at the GOP level for the H.264. The authors use a Master-Worker model and a dynamic scheduling algorithm to distribute the next GOP unit to process to an unused node, and obtain good scalability and speedup. However, this pure GOP-level parallel approach exhibits high latency, and is therefore not applicable to real-time encoding. In [5], Rodriguez et al., propose an approach based on GOP and slice level parallelism. In this approach, the system is composed of homogeneous clusters of workstations. Each cluster receives a GOP unit to process dynamically. A master workstation divides each frame into slices and distributes their processing among workstations. While this approach provides a good trade-off between speedup and latency, this type of encoding can however only be used when the system has access to all the frames of the video sequence to compress, making it unacceptable for real-time telecommunications.

In [8], Chen et al., present a parallel approach at the macroblock level. This algorithm exploits dependencies between macroblocks in a wave front manner. This kind of approach produces a good, but not excellent, speedup. In [9], Steven et al., describe a more efficient approach based on a frame-level and slice-level parallelism. This approach uses five threads on a system with four processors. One thread is used to load the input file (or stream), to save the output file (or stream) and to fill two lists of slices: a priority list of I and P slices and a secondary list of B slices. The other four threads are used to encode slices in parallel. Each thread retrieves and processes the next free slice in the priority list. If this list is empty, the thread retrieves the next free slice in the secondary list. This approach provides excellent speedup, but forces the use of B frames, which is not always possible (for example, the H.264 baseline profile does not allow B frames) and increases latency. Low latency is required in real-time video applications such as videoconferencing.

III. Proposed Multi-Frame and Multi-Slice Parallel Encoding Approach

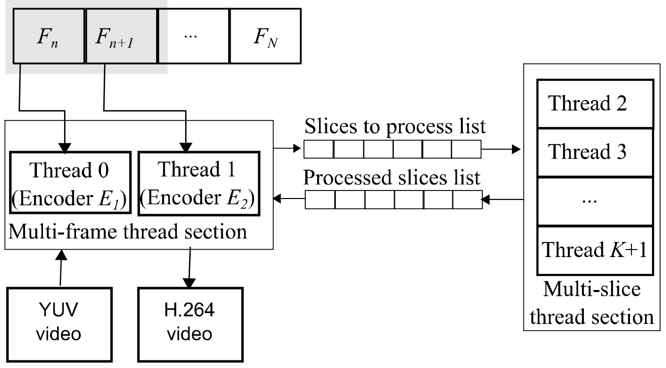

Figure 1. General diagram of the proposed multi-frame and multi-slice parallel approach.

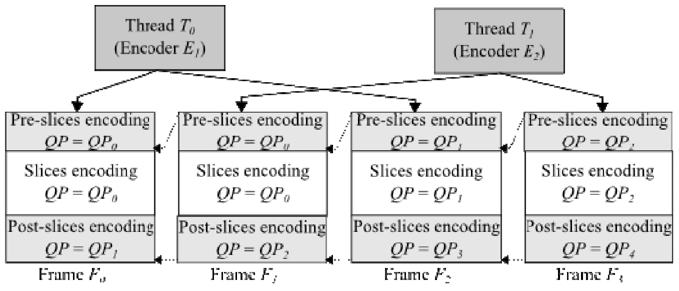
Thread T_0 (Encoder E_1)
Thread T_1 (Encoder E_2)
Pre-slices encoding $QP = QP_0$
Pre-slices encoding $QP = QP_0$
Pre-slices encoding $QP = QP_1$
Pre-slices encoding $QP = QP_2$
Slices encoding $QP = QP_0$
Slices encoding $QP = QP_0$
Slices encoding $QP = QP_1$
Slices encoding $QP = QP_2$
Post-slices encoding $QP = QP_1$
Post-slices encoding $QP = QP_2$
Post-slices encoding $QP = QP_3$
Post-slices encoding $QP = QP_4$
Frame F_0
Frame F_1
Frame F_2
Frame F_3

Figure 2. Processing dependencies and their effect on rate control.

To eliminate the need for B frames in order to efficiently parallelize the encoder, we propose a novel multi-frame and multi-slice (MFMS) parallel approach with three motion estimation modes. Fig. 1 shows a general diagram of the proposed approach for a K cores parallel system.

This approach is composed of two thread sections: a multi-frame thread section and a multi-slice thread section. The multi-frame thread section is composed of two encoders, E_1 and E_2. Each encoder is associated to a unique thread, and is executed simultaneously with the other. Encoder E_1 manages even frames (0, 2, 4, etc.) and encoder E_2 manages odd frames (1, 3, 5, etc.). At any given time, these two encoders must always process neighboring frames (frame F_n and frame F_{n+1}). The multi-slice thread section is composed of a number of threads equal to the number of cores attributed to the application, and as a result, the approach uses more threads than logical processors.

Parallel processing is performed as follows. Each encoder reads (or receives) a YUV frame from an input file (or stream), cuts the frame into rectangular slices of similar sizes (i.e. because a frame is not always divisible into an equal number of MBs lines, some slices may have one more MB line), and fills a list of slices to process. The list is filled according to certain rules described below. Subsequently, these slices are retrieved, encoded and filtered (with the deblocking filter) by threads of the multi-slices threads section. Once processed, a slice is placed in the list of processed slices. When all the slices of a frame are processed, the associated encoder writes these slices in the H.264 output file (or stream) and starts the processing of its next frame.

A. *Processing Dependencies and Rate-Control Delay*

Fig. 2 shows processing dependency relationships between neighboring frames. The processing of each frame is divided into three parts: pre-slice encoding, slice encoding, and post-slice encoding. The pre-slice encoding part performs operations prior to the actual encoding, such as determining the frame type and writing the frame header. The post-slice encoding part performs operations after the actual encoding, such as applying the parallel deblocking filter and updating the rate control status. The slice encoding part encodes the frame's slices and represents the most CPU-intensive part. The pre-slice encoding part of a frame F_{n+1} can only be started when the pre-slice encoding of frame F_n has been completed. Similarly, the post-slice encoding part of a frame F_{n+1} can only be started when frame F_n has been completely processed.

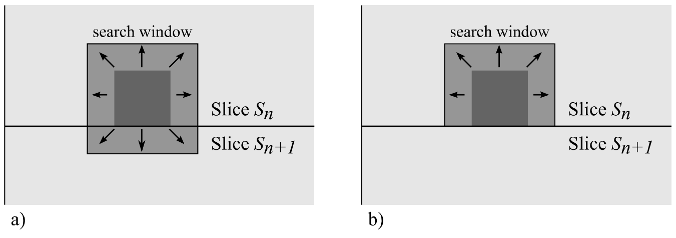

Figure 3. Motion estimation constraints: (a) Unconstrained motion estimation. (b) Motion estimation constrained to the current slice's limits.

The slice encoding part of a frame F_n can only be started after completion of its pre-slice encoding, and depends on the availability of its reference slices (as discussed in section III.C). These dependency relationships have the effect of introducing a delay of one frame in the quantization parameter (QP) determination during encoding. Thus, a frame F_{n+1} applies a quantization equal to QP_n, the QP computed after the encoding of frame F_n, instead of QP_{n+1}. This delay cause rate control to be less precise over short periods (few frames), but does not significantly affect the global video quality and size.

B. Motion Estimation Modes

The proposed approach supports three motion estimation (ME) modes that offer various tradeoffs between speedup and visual quality. Usually, ME is performed inside a square search window (for example, 32 × 32 pixels) of the reference frame. In a multi-slice parallel context, this approach has the disadvantage of forcing the encoder to process a new slice only when all its reference slices (typically three: one on top, one at the same spatial location, and one below) have been processed, and possibly having to wait for them. Our first mode, named Multi Slice Window (MSW), exploits this type of ME window. This is our slowest mode, but the one that provides the best video quality.

To reduce the waiting time, the second mode, named Single Slice Window (SSW), adds an additional constraint to the ME process: ME is performed inside the current slice's limits (as shown in Fig. 3). This mode, which is the fastest, but also the one that affects quality the most, allows an encoder to start processing a slice when its reference slice, located at the same spatial position, becomes available. The quality loss is caused by the reduction of the search window, which reduces the effectiveness of the temporal prediction algorithm (motion estimation). This quality loss is generally greater in a video sequence containing dominant vertical motion, because the motion estimation algorithm cannot track a block that moves from a slice to another one.

The third mode, named Available Slices Window (ASW), is similar to the second mode, but extends the ME to neighboring reference slices, which have already been processed (i.e., once its reference slice is available, the slice will be encoded using all the other reference slices available at that time). This mode is a compromise, in terms of speedup and quality, between the two others modes.

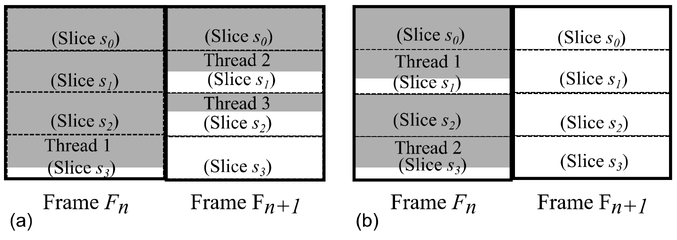

Figure 4. Examples of empty slices to process list:. (a) Single Slice Window (SSW) and Available Slices Window (ASW) modes; (b) Multi Slices Windows (MSW).

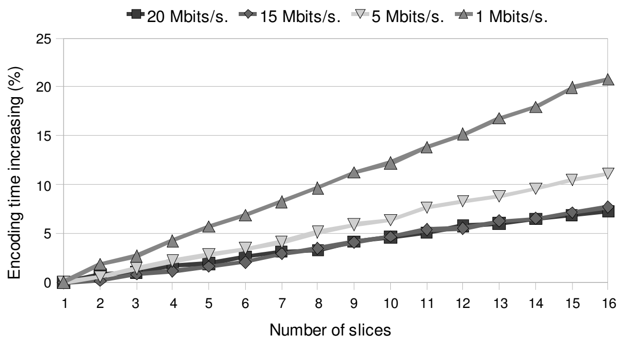

Figure 5. Effet of the number of slices on the encoding time for a sequential (non-parallel) execution on Intel's IPP H.264 video encoder.

C. Selection of the Next Slice to Process and Supplementary Slices

In a pure multi-slice parallel approach, each frame must have a number of slices equal to the number of processor cores to keep all of them as active as possible during the encoding process. However, since some slices are less complex to process than others, workload balancing is not perfect. Adding supplementary slices in this context has the inconvenient of further unbalancing the workload.

In the proposed multi-slice and multi-frame approach, however, adding slices has positive effects on load balancing. When the number of slices equals the number of threads, there will not be enough slices to fill the slices to process list (SPL) to keep all the cores busy at all times. Fig. 4 shows two examples where the SPL is empty: one for the SSW and ASW modes, and one for MSW mode. In these examples, each frame is composed of 4 slices, and 4 threads are used to encode the slices. A white box represents an unprocessed slice, a grey box represents a processed slice and a partially white-grey box represents a slice in process. Fig. 4(a) shows a situation where Thread 4 cannot process Slice 3 in frame F_{n+1} because the processing of Slice 3 in frame F_n is not yet finished. Fig. 4(b) shows a situation where no thread can process a new slice because no slice of frame F_{n+1} has access to all of its reference slices.

To reduce the number of cases where we have an empty slices to process list, we can add supplementary slices. The number of supplementary slices (SS) is defined as the difference between the number of slices per frame and K, the number of threads used to encode slices. The more supplementary slices we use, the fewer cases the number of empty slices to process lists we will have. However, as mentioned earlier, increasing the number of slices reduces the visual quality of the encoded sequence. Moreover, increasing the number of slices slightly increases the complexity of encoding (Fig. 5 shows the effect of the number of slices on the encoding time of a sequential execution on Intel's IPP H.264 video encoder). Therefore, we must add a sufficient number of slices to increase speedup, but not too many, in order to preserve a good video quality.

IV. PERFORMANCE AND DISCUSSION

The proposed approach was implemented based on the H.264 encoder delivered as sample code in the Intel Integrated Performance Primitives (IPP) library, version 6.1.x [2]. Intel's H.264 encoder already comprises a slice-level parallelism approach, which will be compared to our approach.

Our simulations were executed on the first 300 frames of five different YUV video sequences (train-station, horse-cab, sunflower, waterskiing and rally) [10]. These sequences had a resolution of 1280 × 720 pixels at 50 frames per second, and were encoded using the following parameters: logarithm ME method, quarter-pixel motion compensation precision, CABAC entropy encoding, a 16 × 16 pixel ME search area, and constant bit rates (CBR) of 1, 5, 10, 15 and 20 Mbit/s.

Simulations were performed on an HP Proliant ML350 G6 system composed of two quad core Xeon E5530 systems at 2.4 GHz with Hyper-Threading mode turn off. Fig. 6 and Fig. 7 show the average speedup and quality loss compared to the H.264 sequential execution using a single slice per frame. This average was computed for all the above-mentioned video sequences, for a parallel execution using a number of cores varying between 2 and 8. Quality loss was measured in PSNR on the luminance component.

A. Impact of Supplementary Slices on Performance

Fig. 6 shows the impact of adding 0, 1, 2, 3 and 4 supplementary slices on the first ME mode of the proposed parallel approach for a bit rate of 10 Mbit/s. We can observe several important points on this figure. First, the speedup increases with an increase in the number of supplementary slices. Furthermore, the more significant speedup gains are obtained by adding 1 and 2 supplementary slices, respectively. We then notice saturation. The observed gains are more significant when we increase the number of **cores** used. Second, increasing the number of supplementary slices also increases quality loss. Generally, we noticed that adding a number of supplementary slices equal to half the number of cores used was a good choice. Our experiments lead to similar conclusions for the other two modes and for the four others tested bit rates.

B. Comparison Between the Three Proposed Motion Estimation Modes and Intel's Approach

Fig. 7 shows a comparison between the performances of the three proposed ME modes and Intel's multi-slice parallel approach for a bit rate of 10 Mbit/s. For the three proposed modes, we select a number of supplementary slices equal to 4, offering a good trade-off between speed and quality loss for an 8-core parallel execution. For Intel's approach, we select a number of supplementary slices equal to zero, which produces the highest speedup and the lowest quality loss with this approach.

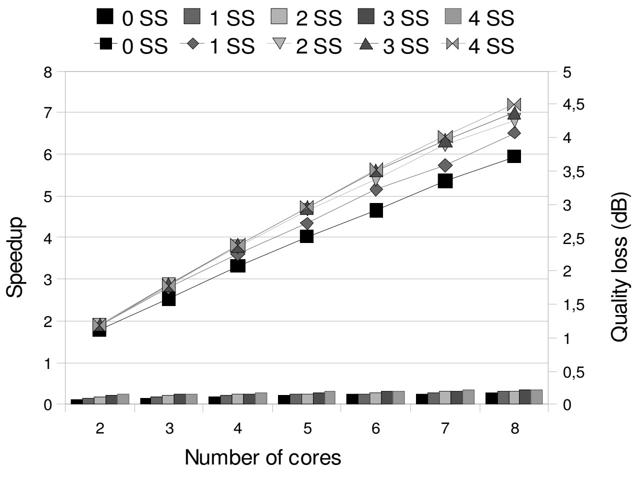

Figure 6. Impact on speedup and quality of the number of supplementary slices (SS) on the MSW mode. Graph show the average results on 5 HD video sequences.

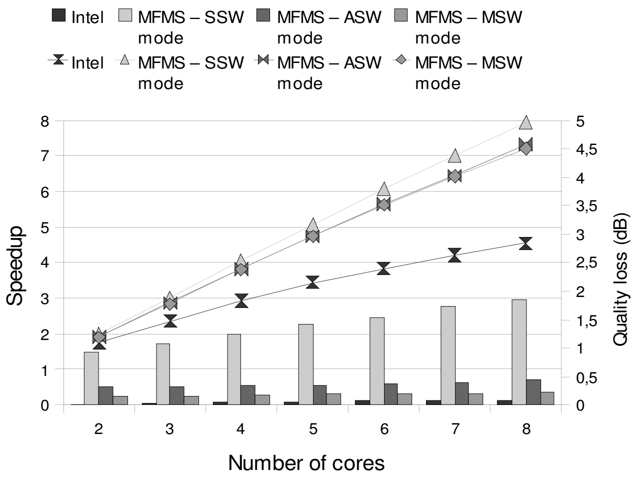

Figure 7. Performance comparison between the three proposed modes with 4 supplementary slices and Intel's approach. Graph show the average results on 5 HD video sequences.

For an 8-core execution, we observed a speedup of 7.95x for SSW mode, the fastest mode, and a speedup of 7.20x for MSW mode, the slowest mode. Therefore, the SSW mode is about 10% faster than the MSW mode. However, the fastest mode yields an average visual quality loss of 1.85 dB, which is high enough to produce a visually perceptible degradation, especially on macroblocks located close to the slice limits, since they use constrained ME. MSW mode produces a quality loss of about 0.22 dB, a barely perceptible loss of quality. Compared to the MSW mode, the ASW mode produces a negligible speedup gain at the expense of a quality loss about twice as high. We therefore suggest using the MSW mode, except when speedup is critical, in which case, SSW may be more appropriate, but at the cost of a much greater quality loss. These recommendations are also applicable for the four others bit rates used for simulations, as results are comparable.

All the modes exceed the speedups achieved by Intel's approach. For example, the SSW mode achieves a speedup of 7.95x; MSW mode, a speedup of 7.20x, and Intel's approach, a speedup of 4.55x, for an 8-core parallel execution. In all cases, the speedup gains are obtained at the cost of increased quality loss, particularly for the SSW mode. However, our MSW mode offers an excellent trade-off between speedup and quality loss that is much more interesting than Intel's approach.

Our approach gets speedups comparable to those obtained by state-of-the-art approaches. Chen et al., [8] and Steven et al., [9] have tested their parallel approach on a 4-processor multi-processor systems. In the first case, the authors obtain a speedup of 3.80x using CIF sequences, and in the second case, they obtain a speedup of 3.95x using CIF sequences. However, this latter approach requires the use of B frames to achieve such a high speedup. By comparison, our fastest mode, the SSW mode, obtains a speedup of 4.02x on HD sequences with 4 cores. This speedup is slightly greater than the expected theoretical acceleration because the restrictions on the ME window reduce the encoding complexity during a parallel execution.

Table I presents the impact of bit rate on performance when MSQ and SSW modes are applied using 8 cores and 4 supplementary slices (for a total of 12 slices). Table I shows that the quality loss decreases, while the speedup increases, with increasing bit rate. The quality loss with MSW mode is entirely due to the usage of slices. As shown in Table I, using 12 slices at 1 Mbit/s produces a quality loss of 0.84 dB (compared to using a single slice as in the sequential execution). With SSW, there is further quality loss due to restricted motion estimation search window.

TABLE I. IMPACT OF BIT RATE ON THE SPEED AND QUALITY FOR THE MSW AND SSW MODES

Bit rate (Mbit/s)	MSW mode		SSW mode	
	Speedup	*Quality loss (dB)*	*Speedup*	*Quality loss (dB)*
1	6.52x	0.84	7.07x	2.64
5	7.06x	0.32	7.77x	2.16
10	7.2x	0.22	7.95x	1.85
15	7.33x	0.18	7.92x	1.67
20	7.35x	0.16	7.99x	1.54

TABLE II. IMPACT OF SEQUENCE ON THE SPEED AND QUALITY FOR THE MSQ AND SSQ MODES WITH A BIT RATE OF 10 MBIT/S.

Sequence	MSW mode		SSW mode	
	Speedup	*Quality loss (dB)*	*Speedup*	*Quality loss (dB)*
Horse-cab	7.36x	0.23	8.15x	3.23
Rally	6.43x	0.46	6.94x	1.6
Sunflower	7.45x	0.06	8.58x	1.62
Tractor	7.45x	0.012	8.15x	1.39
Train-station	7.52x	0.06	8.4x	1.9
Waterskiing	6.98x	0.39	7.45x	1.36
Average	7.2x	0.22	7.95x	1.85

Table II shows the influence of sequence on performance when the MSQ and SSW modes are applied with 8 cores and 4 SS. Speedup and quality decrease on sequences with high vertical motion such as rally. As observed earlier, SSW can obtain speedups greater than the expected theoretical acceleration because the restrictions on the ME window reduce the encoding complexity during a parallel execution.

V. CONCLUSION

In this paper, we presented a new multi-frame and multi-slice parallel video coding approach applied to H.264. This approach achieves high speedup, low latency and low video quality loss, without requiring the use of B frames. Our approach produces the highest speedups compared to those in the literature which are not requiring B frames. Our speedups are also comparable to those obtained by the best methods using B frames. Not requiring B frames makes our approach usable in a real-time video transmission context such as video-conferencing. Our high speedups are obtained from supporting a number slices equal or greater than the number of cores used. The flexibility we permit in the number of slices used is important because some application need a high number of slices for different reasons: maximum packet size, error resilience, parallel decoding, etc. Furthermore, our approach can be implemented with little effort in an H.264 encoder supporting a multi-slice parallel approach, such as Intel's. The high speedup of the approach allows the encoding of HD video sequences in real time on recent multicore systems.

ACKNOWLEDGMENT

This work was funded by Vantrix Corporation and by the Natural Sciences and Engineering Research Council of Canada under the Collaborative Research and Development Program (NSERC-CRD 326637-05).

REFERENCES

[1] ISO/IEC 14496-10 and ITU-T Rec, "H.264, Advanced video coding for generic audiovisual services," ed, 2003.

[2] Intel. (2010, *Intel® Integrated Performance Primitives 6.1 – Code Samples* Available: http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-code-samples/

[3] S. Chien, *et al.*, "Hardware architecture design of video compression for multimedia communication systems," *IEEE Communications Magazine,* vol. 43, p. 123, 2005.

[4] L. Changying, *et al.*, "The Research of H.264/AVC Video Encoding Parallel Algorithm," in *Intelligent Information Technology Application, 2008. IITA '08. Second International Symposium on*, 2008, pp. 201-205.

[5] A. Rodriguez, *et al.*, "Hierarchical Parallelization of an H.264/AVC Video Encoder," in *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, 2006, pp. 363-368.

[6] S. Shu-wei and C. Shu-ming, "An Efficient Parallel Algorithm for H. 264/AVC Encoder [J]," *Acta Electronica Sinica,* vol. 2, 2009.

[7] H. K. Zrida, *et al.*, "High level H. 264/AVC video encoder parallelization for multiprocessor implementation," 2009, pp. 940-945.

[8] Y. Chen, *et al.*, "Implementation of H. 264 encoder and decoder on personal computers," *Journal of Visual Communication and Image Representation,* vol. 17, pp. 509-532, 2006.

[9] G. Steven, *et al.*, "Efficient multithreading implementation of H.264 encoder on Intel hyper-threading architectures," in *Information, Communications and Signal Processin and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003*, pp. 469-473 Vol.1.

[10] Xiph.org. 2011, *Test Media*. Available: http://media.xiph.org/video/derf/