Parallel Deblocking Filtering in MPEG-4 AVC/H.264 on Massively-Parallel Architectures

Bart Pieters, Charles-Frederik J. Hollemeersch, Jan De Cock, *Member, IEEE,* Peter Lambert, *Member, IEEE,* Wesley De Neve, and Rik Van de Walle, *Member, IEEE.*

***Abstract*—The deblocking filter in the MPEG-4 AVC/H.264 standard is computationally complex because of its high content adaptivity, resulting in a significant number of data dependencies. These data dependencies interfere with parallel filtering of multiple macroblocks on massively-parallel architectures. In this paper, we introduce a novel macroblock partitioning scheme for concurrent deblocking in the MPEG-4 AVC/H.264 standard, based on our idea of Deblocking Filter Independency, a corrected version of the Limited Error Propagation Effect proposed in the literature. Our proposed scheme enables concurrent macroblock deblocking of luma samples with limited synchronization effort, independently of slice configuration, and is compliant with the MPEG-4 H.264/AVC standard. We implemented the method on the massively-parallel architecture of the Graphics Processing Unit (GPU). Experimental results show that our GPU implementation achieves faster-than real-time deblocking at 1309 frames per second for 1080p video pictures. Both software-based deblocking filters and state-of-the-art GPU-enabled algorithms are outperformed in terms of speed by factors up to 10.2 and 19.5 respectively for 1080p video pictures.**

***Index Terms*—deblocking, GPU, MPEG-4 AVC/H.264, in-loop filtering, massively-parallel**

I. Introduction

THE in-loop deblocking filter in the MPEG-4 AVC/H.264 video coding standard [1] is designed to reduce blocking artifacts caused by quantization. The filter is highly content-adaptive, resulting in increased filter efficiency, but also in increased computational complexity [2]. This computational complexity is mainly due to the conditional processing of block edges and the interdependencies of successive filtering steps. Edge filtering modifies samples by complex filters using up to five taps. These can occur over slice and macroblock boundaries, introducing dependencies between filtered edges which interfere with parallel execution. Therefore, most deblocking algorithms proposed in the literature are aimed at pipelined [3] or serial [4]–[6] processing of macroblocks.

Manuscript received March 16, 2010; revised May 20, 2010. Accepted for publication July 24, 2010.

Copyright (c) 2010 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending an email to pubs-permissions@ieee.org.

The research in this paper was funded by Ghent University, the IBBT, the IWT, FWO-Flanders, BFSPO, and the European Union.

All authors but Wesley De Neve are with Multimedia Lab, ELIS, Ghent University — IBBT, Belgium (e-mail: Bart.Pieters, CharlesFrederik.Hollemeersch, Jan.DeCock, Peter.Lambert, Rik.VandeWalle@ugent.be). Wesley De Neve is with the Image and Video Systems Lab, Department of Electrical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Republic of Korea (e-mail: Wesley.DeNeve@kaist.ac.kr).

In this paper, a novel parallel processing algorithm for the deblocking filter in MPEG-4 AVC/H.264 is presented, facilitating concurrent filtering of the luma component of macroblocks on the massively-parallel architecture of the GPU, while staying compliant with the standard. Specifically, we propose a newly discovered macroblock independency that is denoted as Deblocking Filter Independency (DFI). It is based on our corrected and improved version of the Limited Error Propagation Effect introduced by Wang *et al.* in [8]. By removing the inaccuracies for intra-coded slices that caused lossy filtering by Wang *et al.*, correct filtering results according to the MPEG-4 AVC/H.264 standard can be achieved. Next, a novel macroblock partitioning scheme is presented making use of our idea of DFI, enabling parallel processing at macroblock level with limited synchronization and no recalculation overhead. The proposed scheme is implemented on the massively-parallel architecture of the GPU using the NVIDIA CUDA platform [7] and is evaluated.

The remainder of this paper is organized as follows. Section II briefly describes the in-loop deblocking filter as defined in the MPEG-4 AVC/H.264 standard and approaches to parallel deblocking in the literature are analyzed. Section III proposes the DFI while Section IV introduces our novel macroblock partitioning scheme to enable concurrent filtering using the DFI. Section V subsequently shows the experimental results of our implementation and a number of comparisons and is followed by our conclusions in Section VI.

II. Deblocking Filtering in the MPEG-4 AVC/H.264 Design and Related Work

With the in-loop deblocking filter, each macroblock is filtered in raster-scan order with optional filtering over slice boundaries. The edge filtering order for the luma component in the current macroblock is shown in Fig. 1, starting with filtering four vertical edges, followed by filtering four horizontal edges. The evaluation of each filter for the luma component may require up to four samples (p_{0-3}, q_{0-3}), whereas each filter may update up to three samples (p_{0-2}, q_{0-2}) on both sides of an edge. This makes the deblocking filter an in-place filter as the filtered sample values are used in the filtering of the next edge. All of the edges in Fig. 1 are conditionally filtered based on a Boundary-Strength (BS) parameter and the sample gradient across the boundary. The BS parameter is calculated using information about quantization parameters (QPs), coded residuals, motion vectors, and reference frames of the current and adjacent blocks. In case of intra coding, the BS parameter

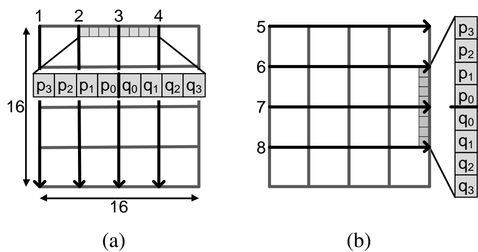

Fig. 1. Filtering of luma macroblock edges. (a): horizontal filtering of vertical edges, (b): vertical filtering of horizontal edges.

can be equal to 4 at macroblock edges, indicating the use of a strong filter that filters four samples (p_{0-3} and q_{0-3}) at each side of a macroblock edge. The resulting three samples p_{0-2} and q_{0-2} are written back at their side of the edge. When the BS parameter is equal to 3, 2, or 1, the normal filter is selected reading three samples at each side of the edge, p_{0-2} and q_{0-2}, and filtering samples p_{0-1} and q_{0-1}. Two additional threshold functions are used to determine whether each set of samples from both sides of the edge need filtering. These functions are dependent on the QP and offset values conveyed within the slice header as well as samples p_{0-1} and q_{0-1}. Finally, when the BS parameter is equal to 0, the edge is not filtered.

As stated in the literature [4]–[6], [11], both BS parameter calculations and chroma edge filtering can be executed in parallel on a macroblock basis because of limited interdependencies. Luma deblocking however shows a high number of interdependencies between filtered luma samples: filtered samples are used in subsequent filtering steps and filtering also occurs across macroblock and slice boundaries. Therefore, several techniques for deblocking rely on sequential calculations [4]–[6]. Conventional parallel approaches involve parallel processing of rows or columns of samples in a macroblock, either with a pipelined design [3] or without a pipelined design [4], [11], [12]. As macroblocks measure 16 by 16 luma samples, a maximum of 16 sample rows or columns can be processed concurrently. One suggested method to increase parallelism is the use of wavefront techniques [4], [11], [12]. However, only a limited amount of parallel processing is possible as the number of macroblocks per wave vary. Additionally, these techniques require a high number of synchronization points, i.e. one per wave. On many massively-parallel architectures, the overhead associated with a synchronization point decreases the performance gain benefit from parallel processing.

Wang *et al.* [8] have shown that not all samples of a given macroblock are dependent on previously-filtered samples. As previously stated, the strong filter may alter three samples at each side of macroblock edge S: p_2^S, p_1^S, p_0^S, and q_2^S, q_1^S, q_0^S as illustrated in Fig. 2. The next normal filter is dependent on these filtered samples, however, only partially. The last filtered sample q_1^N is only dependent on q_3^S of the previously-filtered edge. This sample is unaffected by the strong filter and therefore an original unfiltered sample. Further, the result of the next normal filter only depends on sample q_1^N, the filtered sample independent of the strong filter. Consequently, this implies that all samples starting from q_1^N are supposed to be independent of the previously-filtered macroblock. Wang *et al.* suggest a parallel technique based on this finding. Specifically, they propose to divide a video picture in a number of rectangular parts, either horizontally or vertically, in which macroblocks are processed in raster-scan order. Left or top edges of a part are filtered incorrectly because of macroblock sample interdependencies, introducing an error. However, this error propagates only for a limited amount of samples because of the observed limited dependencies. Wang *et al.* call this the *Limited Error Propagation Effect*. After processing all rectangular parts, the left and top edges of the parts are recalculated using the correctly filtered samples from the adjacent part, thus correcting the previously-introduced errors. With this technique, synchronization is minimized but concurrency is not maximized.

III. Proposed Deblocking Filter Independency

Wang *et al.* introduced the notion of Limited Error Propagation in [8]. The authors however ignore the fact that filtering of the second block edge only occurs when the threshold function using p_0 and p_1 is true as shown in Fig. 2 and as described in the standard. These sample values, here p_1^N and p_0^N, make q_1^N dependent on p_0^N and p_1^N. As p_1^N is the previously-filtered value q_2^S, the sample q_1^N, claimed independently filtered is still dependent on the filtered results by the strong filter. Further examining q_2^S, we see that the value of this sample is calculated using samples as far as p_0 of the strong filter (column p' in Fig. 2), a sample filtered in the previous macroblock. This implies that the sample claimed independently filtered, is actually dependent on previously-filtered macroblock samples. When the normal filter is used to process the macroblock edge instead of the strong filter, q_2^S is unaffected by the filter. Therefore, p_0^N and p_1^N can be evaluated using unfiltered values and the decision of filtering the second block edge can be made correctly. Consequently, this observed dependency makes it impossible to filter samples of column f to n in Fig. 2 in parallel over different macroblocks when the strong filter is used. Therefore, the method proposed by Wang *et al.* does not provide correct results when deblocking video picture rectangles in parallel. We simulated results of the method of Wang *et al.* showing errors introduced in columns f-i causing errors in the decoded output. This shows that the method of Wang *et al.* is not compliant with the standard.

We propose a necessary modification of the idea of Limited Error Propagation to enable correct filtering on parallel architectures according to the MPEG-4 AVC/H.264 standard. This modification is based on what we call the Deblocking Filter Independency (DFI), which is also visualized in Fig. 2. When the third block edge between column h and i is investigated, we notice that for this edge the decision to filter the edge can be evaluated without additional dependencies. This is because the p_0^M and p_1^M used are samples unaffected from the normal filter executed on the second block edge. Whether q_1^M is filtered or not is independent from previously-filtered values – q_1^M only depends on unfiltered samples p_0^M, q_0^M, q_1^M, and q_2^M. Therefore, all samples starting from q_1^M, or column j can be filtered independently on a macroblock basis.

Because the deblocking filter in MPEG-4 AVC/H.264 is a two-dimensional filter, we studied the DFI in two directions,

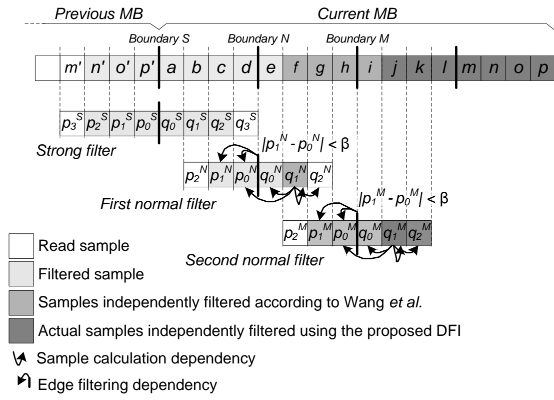

Fig. 2. Limited Error Propagation Effect proposed by Wang *et al.* and the proposed Deblocking Filter Independency.

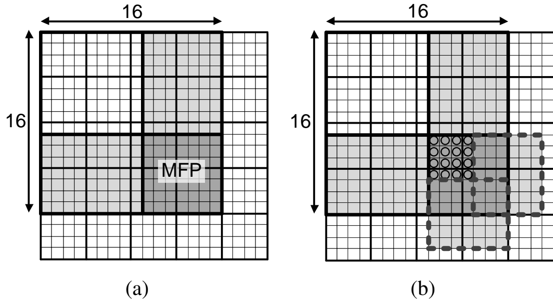

Fig. 3. The DFI in two dimensions. Light-gray: vertical or horizontal, dark-gray: vertical and horizontal, circled: definitive state. (a): DFI within a macroblock; (b): after filtering of neighboring macroblocks.

shown in Fig. 3(a). In the figure, the bottom and right light-gray areas represent samples independently filtered from the adjacent upper and left macroblock respectively. The two areas in Fig. 3(a) overlap. This overlapping part contains samples where the filtering process can start independently from the left and upper macroblock and can therefore be filtered for all macroblocks in parallel. In this paper, these filtered parts of a macroblock are called *Macroblock Filter Partitions* (MFPs). Essentially, MFPs are spatial clusters of samples that after macroblock-parallel filtering of their corresponding edges go into the same filtered state as if filtered by the raster-scan order algorithm at a given time. For a CIF video picture, an MFP would consist out of 396 sample clusters, one for each macroblock. However, only part of the MFP shown in Fig. 3(a) contains definite correctly-filtered values. Indeed, Fig. 3(b) shows the influence of filtering of the macroblock to the right and bottom of the current macroblock when filtering in raster-scan order. It is possible that a strong filter is used for edge 1 (see Fig. 1) of the macroblock to the right and for edge 5 of the macroblock to the bottom of the current macroblock, effectively influencing part of the overlapped area. The samples of the MFP that cannot be influenced are therefore in their final filtering state, showed in Fig. 3(b) circled. If macroblocks were to be filtered independently of adjacent filtered macroblocks, only this small part of the MFP would be filtered correctly. Therefore, in order to filter an entire macroblock correctly, a specific processing order of MFPs is required.

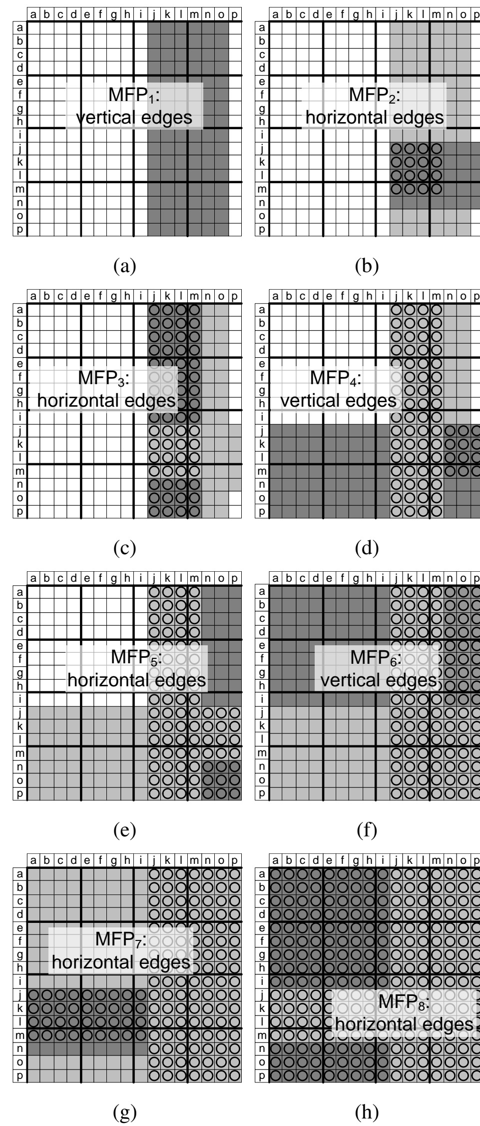

Fig. 4. Proposed macroblock partitioning over successive passes. White: unfiltered samples, light-gray: previously-filtered samples, dark-gray: filtered in current pass, circled: samples in their final state.

IV. Proposed Macroblock Partitioning Scheme for Macroblock-Parallel Processing

By partitioning a macroblock, data-independent samples are gathered together, forming an MFP. Samples of this MFP, together with original unfiltered samples, can then act as the basis for filtering of samples of the next MFP. When an MFP is based on the filtered output of another MFP, a synchronization point is introduced as processing of the previous MFP must be finished for all macroblocks in the picture. MFPs of subsequent steps are numbered, with MFP_n based on original unfiltered samples and samples of $\text{MFP}_{\text{m,m<n}}$.

Fig. 4 shows the proposed partitioning scheme. The entire video picture is divided into eight different MFPs, independent of slice configuration. We call filtering of an MFP with a subsequent synchronization point a *pass*. Each sub picture shows the effect of a pass on a single macroblock of the video picture after filtering the MFP. The effects of filtering of surrounding

macroblocks in the same MFP on the current macroblock is also indicated on each sub figure. We distinguish five types of samples in Fig. 4: white, unfiltered samples; light-gray, previously-filtered samples; dark-gray, samples filtered in the current pass; circled, filtered samples in their final state.

Filtering starts with the first two passes, representing the discussed case in Fig. 3. Vertical edges 3 (partially) and 4 are filtered in pass 1 (Fig. 4(a)) because of the DFI stating that these columns are independent of previously-filtered macroblocks. The filtered results (MFP_1) are subsequently used for the filtering of horizontal edges 7 and 8 in pass 2 (Fig. 4(b)), again according to the DFI. As stated before, part of this MFP already contains filtered samples in their final state, shown circled in Fig. 4(b). In pass 3, samples in MFP_3 are filtered and written. Here, the top macroblock edge is possibly filtered by the strong filter, influencing up to three rows (n, o, p) of samples at the bottom of the macroblock above the macroblock in question. This is shown by the three rows of influenced samples at the bottom of Fig. 4(c). Note that part of row n is filtered for the third time. This corresponds with the filtering of column n' in Fig. 2. The resulting filtered samples are found correct and in their final state because the filtering of horizontal edges in MFP_3 only depends on correctly filtered samples from MFP_2 of the macroblock above. Note that filtering of the bottom-left samples of the macroblock located to the top-right of the current macroblock prohibits us to filter columns n, o, and p. The next pass filters samples of vertical edges starting from row j in MFP_4 (Fig. 4(d)). The DFI states that these samples can be filtered independently from the macroblock above the current macroblock. Results of MFP_4 depend on up to four columns of samples from the macroblock to the left, i.e. filtered results from $MFP_{1\text{-}2}$. Next, the remaining horizontal edges in columns n-p are filtered in pass 5 (Fig. 4(f)) as part of MFP_5, making it possible to filter vertical edges 1 to 3 in pass 6 (Fig. 4(f)). Now the process can be repeated again for MFP_7 (Fig. 4(g)) because of the DFI and the presence of correctly-filtered samples for horizontal edges 7 and 8. Finally, all remaining horizontal edge samples based on samples of MFP_7 can be filtered in pass 8 (Fig. 4(h)).

Looking at MFP_2 and MFP_3 of Fig. 4, it is clear that these MFPs can be merged into one MFP as horizontal edges are filtered from top to bottom. Likewise, MFP_7 and MFP_8 can be merged. Hence our partitioning scheme allows an entire video picture to be filtered using six synchronization points. This way, parallel execution benefits outweigh synchronization overhead. Concurrency over macroblocks is maximized as all macroblocks in each MFP of the video picture can be filtered in parallel, independent of slice configuration.

V. Experimental Results and Discussion

We implemented our proposed macroblock-parallel deblocking algorithm next to the state-of-the-art wavefront algorithm and a serial method on the massively-parallel architecture of the GPU using the NVIDIA CUDA platform (version 3.0b1). We compared the GPU implementations to the highly-optimized CPU filter present in libavcodec [10]. Both luma and chroma component filtering was measured. For the performance tests, progressive video sequences of different resolutions were used with YUV4:2:0 sampling and for each resolution, results for three video sequences were averaged. Each video picture consisted of one or four intra-coded slices using either a QP of 27 or 45. Performance tests were done on a system running Windows 7, an AMD Q9950 CPU and an NVIDIA 8800GTX (G80), GTX280 (G200), and GTX480 (GF100) graphics card. These cards have respectively 128, 240, and 480 Stream Processors (SPs). Output of the GPU-enabled filters was compared to that of the reference decoder and was found bit-accurate. Indeed, as mentioned before, the proposed MB-parallel filter deblocks according to the MPEG-4 AVC/H.264 specification and therefore introduces no error.

Three scenarios were simulated, i.e., a decoder scenario and two encoder scenarios encoding video pictures using an I and IPPP GOP structure. For the decoder scenario, scenario 1, the bitstream is uploaded to the GPU, simulating decoding on the GPU. As all information is resident in GPU memory, there is no synchronization required between GPU and CPU. Our measurements showed that GOP structure and QP have only limited influence on processing speed for the GPU algorithms because of the use of predicated execution instructions for filtering. Therefore, we included results for the worst-case QP and slice configuration for these algorithms. Table I compares our proposed GPU algorithm to the GPU wavefront and serial filtering method, showing the number of frames filtered per second (frames per second; fps). It can be seen how our proposed MB-parallel method outperforms both serial and wavefront methods by a factor of 187.0 and 19.5 respectively for 1080p on the GF100. Both serial and wavefront filters show consistently low fps as they require a high number of synchronization points and exploit limited parallelism. Our algorithm minimizes synchronization points and maximizes parallelism. The table clearly shows how performance scales linearly with the amount of streaming processors for the proposed MB-parallel method. For example, the GF100 (480 SPs) filters with a factor of 2.1 faster than the G200 (240 SPs) for 1080p. This is not the case for small resolutions where there are not enough edges available to be filtered on each SP.

Next, we discuss results for scenarios 2 and 3 in Table I where an encoder compresses video pictures using an I and IPPP GOP structure respectively. These scenarios require information processed by the encoder, such as quantization parameters, reconstructed picture, etc. to be uploaded in GPU memory. The communication cost required to provide the GPU with input data was included in the measurements, as well as the time needed to download a deblocked picture to system memory. In case of an I GOP structure, perfect pipelining of GPU and CPU communications is possible as encoding of the next frame can start before deblocking of the previous ends. Starting from the G200, communication costs can be hidden by kernel execution. Furthermore, GPU performance is limited by the communication speed between GPU and system memory. Our measurements show this to be true for the GF100, leaving some of its streaming processors idle. Note that results for the G200 for scenario 1 and 2 converge for high resolutions as for these video sequences, kernel execution time outweighs transfer speed. For an IPPP GOP structure, perfect pipelining

TABLE I
EXPERIMENTAL RESULTS FOR CPU AND GPU IMPLEMENTATIONS, INCLUDING THE PROPOSED MB-PARALLEL APPROACH (IN FRAMES PER SECOND) USING AN AMD Q9950 CPU AND GEFORCE 8800GTX (G80), GTX240 (G200), AND GTX480 (GF100) GPUS.

Scenario	Resolution	CPU				GPU								
		libavcodec				Serial			Wavefront			Proposed MB-parallel		
		Single-Core		Quad-Core		G80	G200	GF100	G80	G200	GF100	G80	G200	GF100
		QP27	QP45	QP27	QP45									
1	CIF	2501	3096	5409	5335	90	92	143	133	173	509	1984	2892	4762
1	480p	615	815	2091	2472	24	25	42	44	93	261	948	2129	4403
1	720p	279	390	949	1167	10	10	15	24	55	147	596	1204	2632
1	1080p	76	81	258	275	4	4	7	10	35	67	267	637	1309
2	CIF	1591	1569	5409	5335	90	92	143	129	173	504	1567	2588	2811
2	480p	499	474	1697	1612	22	25	42	42	92	262	668	1531	2702
2	720p	185	178	618	587	8	10	15	23	55	146	332	1201	1599
2	1080p	76	75	254	248	4	4	7	9	35	66	154	622	762
3	CIF	2501	3096	8503	10012	89	92	141	129	168	491	1443	2375	2717
3	480p	615	815	2091	2472	22	24	42	42	89	260	600	1472	1748
3	720p	279	390	949	1167	8	10	14	22	52	133	310	724	860
3	1080p	76	81	258	275	4	4	7	9	33	66	138	351	412

is not possible and CPU and GPU must synchronize execution causing the GPU to idle.

For scenarios 2 and 3, we compared the proposed MB-parallel method with the highly-optimized deblocking implementation in libavcodec. The table shows our proposed method to outperform all CPU-based methods for both scenarios for high-definition resolutions. For example, the GF100 establishes a speedup factor of 10.2 for 1080p over a single CPU core for scenario 2. For comparisons with a multi-core version of libavcodec, we disabled deblocking over slice boundaries and used video sequences with four slices. In scenario 2, our method shows a speedup of 3.0 compared to four CPU cores. The table shows how performance of the proposed algorithm lowers from 762 fps in scenario 2 to 412 fps in scenario 3 for 1080p sequences as the transfer and synchronization overhead has increased. Furthermore, as less filtering is used with inter-coded slices, performance of the CPU filter increases from 248 to 259 fps, causing GPU and CPU speed to converge. For small resolutions, the GPU method is outperformed as there are no communication costs for the CPU method and the filtered image fits entirely in the CPU cache. The MB-parallel implementation outperforms the multi-core version for the highest resolution with a speedup factor of 1.7.

VI. CONCLUSION

In this paper, we presented a novel parallel processing algorithm for the deblocking filter in the MPEG-4 AVC/H.264 standard, enabling concurrent filtering of macroblocks. We showed that the level of parallelism of state-of-the-art parallel deblocking algorithms is insufficient and the number of synchronization points too high for use on massively-parallel architectures. Therefore, a novel macroblock partitioning algorithm was introduced, based on our corrected version of the Limited Error Propagation Effect, that is compliant with the MPEG-4 AVC/H.264 standard. It allows maximum parallel processing concurrency for deblocking of video pictures on the macroblock level, independent of slice configuration while requiring only six synchronization points. The proposed parallel technique was tested on the massively-parallel architecture of the GPU and implemented using the NVIDIA CUDA platform. Experimental results show that our deblocking method and its implementation allow for faster than real-time deblocking at 1309 frames per second for 1080p video pictures on a GPU. In particular, our implementation outperforms both an optimized CPU-based filter and state-of-the-art parallel GPU methods in terms of speed by a factor up to 10.2 and 19.5 respectively, limited by the system bus communication overhead in today's computer systems.

REFERENCES

[1] Joint Video Team (JVT) of ITU-T and ISO/IEC JTC 1, "Advanced Video Coding for Generic Audiovisual Services," ITU-T Rec. H.264 and ISO/IEC 14496-10 (MPEG-4 AVC), Version 5, July 2007.

[2] P. List, A. Joch, J. Lainema, G. Bjøntegaard, and M. Karczewicz, "Adaptive Deblocking Filter," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 7, pp. 614–619, July 2003.

[3] K. Xu and C. Choy, "A Five-Stage Pipeline, 204 Cycles/MB, Single-Port SRAM-Based Deblocking Filter for H.264/AVC," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 18, no. 3, pp. 363–374, March 2008.

[4] Z. Zhao and P. Liang, "Data Partition for Wavefront Parallelization of H.264 Video Encoder," in *IEEE International Symposium on Circuits and Systems*, May 2006.

[5] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer, "Efficient Parallelization of H.264 Decoding with Macro Block Level Scheduling," in *IEEE International Conference on Multimedia and Expo*, July 2007, pp. 1874–1877.

[6] G. Amit and A. Pinhas, "Real-Time H.264 Encoding by Thread-Level Parallelism: Gains and Pitfalls." in *IASTED PDCS*, September 2005, pp. 254–259.

[7] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide Version 2.0*, NVIDIA Corporation, July 2008.

[8] S.-W. Wang, S.-S. Yang, H.-M. Chen, C.-L. Yang, and J.-L. Wu, "A multi-core architecture based parallel framework for H.264/AVC deblocking filters," *Signal Processing Systems*, vol. 57, no. 2, pp. 195–211, 2009.

[9] G. J. Sullivan and T. Wiegand, "Video Compression - From Concepts to the H.264/AVC Standard," *Proc. the IEEE, Special Issue on Advances in Video Coding and Delivery*, vol. 93, no. 1, pp. 18–31, January 2005.

[10] *libavcodec*, Part of FFMPEG, version 1.0, July 2007. [Online]. Available: http://ffmpeg.mplayerhq.hu/

[11] J. C. A. Baeza, W. Chen, E. Christoffersen, D. Dinu, and B. Friemel, "Real-Time High Definition H.264 Video Decode Using the Xbox 360 GPU," in *Proc. of SPIE: Applications of Digital Image Processing XXX*, vol. 6696, no. 1, 2007.

[12] F. H. Seitner, M. Bleyer, M. Gelautz, and R. M. Beuschel, "Evaluation of data-parallel H.264 decoding approaches for strongly resource-restricted architectures," *Multimedia Tools and Applications*, pp. 1380–7501, 2009.