Analyzing Scalability of Deblocking Filter of H.264 via TLP exploitation in a new many-core architecture

Abstract

In this paper we present results of parallelization of Deblocking Filter (DF) of H.264 video codec on Decoupled Threaded Architecture (DTA). We parallelized the code trying to exploit all available thread level parallelism and to make it suitable for DTA architecture. Experimental results show that significant speedup can be achieved and that DTA architecture can efficiently exploit available parallelism. We also show comparison with parallelized version of DF for Cell architecture.

Track area: DSD Digital System Design
Conference topic area: T6
Keywords: H.264, deblocking filter, TLP, many-core, scalability, DTA

1. Introduction

Today's multimedia systems demand more and more computational power since the quality of content that they provide is improving. In particular, users show constant demand for videos with higher resolution even on mobile devices. H.264, also known as MPEG4 part 10 or MPEG-4 AVC (Advanced Video Coding) is a video coding standard aimed at providing high video quality even at lower bitrates. It was developed with many application fields in mind, such as high resolution video (for satellite, cable or DSL broadcast), video storage (HD-DVD, blu-ray disc), and internet and multimedia telephony systems [1].

Current single core architectures performance cannot keep up with growing requirements for computational power. Since the technology has enabled putting more resources on a single chip, it is now possible to use many-core processors even in embedded devices. The many-core architecture that we are developing, the Decoupled Threaded Architecture (DTA) [2], is based on a coarse-grained dataflow among threads, and on their non-blocking execution. It also exploits distribution of processing elements to overcome wire delay problem and to improve the overall performance.

One more example of many-core architecture is a new research chip from Intel that contains 80 simple cores, where each core contains two programmable floating point engines. Each core contains a 5-point message passing router, and is connected to other cores in a 2D mesh network. Unlike DTA, this chip exploits standard programming model.

TRIPS [3] is another example of many core architecture that uses “medium size” tiles that can be configured either as processing elements, memory, cache or registers. While DTA exploits dataflow execution at the thread level and control-flow inside one thread, TRIPS does the opposite. Indeed, TRIPS executes hyper-blocks in a control-flow order, and inside these blocks execution is dataflow.

Cell Broadband Engine Architecture (CBEA) [4] combines one Power Architecture core with SIMD processing elements that are called SPEs (Synergetic Processing Elements). In the current implementation, one CBEA processor has 8 SPEs that are interconnected by a circular ring with four channels. The main difference between CBEA and DTA is the programming model that is used.

Many-core architectures have become widely used. Therefore, parallelization of programs that are used for providing multimedia content, such as video codecs, and running them on many-core processors is a promising way to improve the performance. In our work we have focused on parallelizing Deblocking Filter (DF) of the H.264 codec, and on utilizing the advantages that DTA offers to exploit available Thread Level Parallelism (TLP). We chose DF because it is one of the most time consuming part of the code [5] [6].

The rest of the paper is organized as follows. Section 2 provides a high-level overview of H.264 deblocking filter and its parallelization possibilities. Section 3 explains the basics of DTA architecture and DF implementation for it. Section 4 presents obtained results on the DTA architecture and comparison with Cell. Conclusions are shown in Section 5.

2. Deblocking Filter of H.264

Encoding and decoding process in H.264 audio/video codec is composed of several different steps. Deblocking filter is one of the steps in the process. By profiling of H.264 it can be seen that deblocking filter consumes about 7% of total decoder processing time [5]. In the case when using Altivec [7] extensions for optimizing H.264 kernels for PowerPC and leaving deblocking filter non optimized, deblocking filter portion of execution time of H.264 decoder can grow up to 49% [6]. Obviously, deblocking filter consumes significant portion of the decoder, both with and without optimizations, and therefore it is important to execute it as efficiently as possible.

Steps in H.264 operate on macroblocks (MBs), which are blocks of 16x16 pixels. Because decoding process is block-based, sharp edges may appear between the blocks after discrete cosine transformation (DCT) is applied. This is known as “blocking”. The purpose of having a deblocking filter is to try to eliminate these artifacts by smoothing the edges of adjacent blocks. In H.263 version of the standard deblocking filter was optional step, but from H.264 it is a part of the standard.

Here we will give just a rough idea of a deblocking filter process, for more information refer to [8]. Deblocking filter basically modifies pixels at the edges of macroblocks in cases when they meet certain conditions. The type of modification that is performed depends on the parameter called boundary strength and it varies on the macroblock type and coding conditions. In deblocking filter, macroblock (MB) processing is done on the level of even smaller blocks of 4x4 pixels [9]. Filtering process is done on both vertical and horizontal edges of blocks. It starts at the left vertical edge and proceeds at all internal edges. After filtering is done for vertical edges, it is repeated for horizontal edges starting from the top. Filtering is done for all three color components independently.

There are several possibilities to exploit thread level parallelism in the deblocking filter [10]. At the MB level, all MBs that don’t have dependencies between them can be processed at the same time. One MB can’t be processed before MBs on its left and above it have already been processed (other steps in H.264 introduce additional dependencies, but here we analyze just DF). For example, frame in CIF resolution of 320x240 pixels, which has 300 MBs, can be processed in total of 34 time slots. Maximal number of MBs that can be processed in parallel is 15 and it lasts for 6 time slots. However, average number of available independent MBs is 8.82, and it is available for more than 50% of execution time. In higher resolution the number of MBs increases. For example, in FHD resolution (1920x1088), maximal number of independent MBs is 68 (average 43,64) and it is available for 57 out of 187 time slots.

Next, all three color components can be processed in parallel (Y, Cb and Cr). One more opportunity for parallelism is to process 4x4 blocks in parallel. At each step in both vertical and horizontal pass, 4 of these blocks are processed. It is practically data level parallelism, but it can be transformed in thread level by processing each of 4 blocks in a separate thread. This is done by unrolling appropriate loops in the code.

3. A new many-core architecture: DTA

DTA [2] is based on SDF execution paradigm [11]. DTA utilizes threads that are non-blocking and memory accesses are decoupled from execution. Threads communicate among them in a producer-consumer fashion, and a thread will start its execution only when all its data is ready in local (frame) memory. Processing elements (PE) in DTA are grouped into nodes (Figure 1), where dimension of each node is determined with a constraint that each PE must be reachable in one cycle. On the other hand, communication among nodes is slower, and interconnection network is more complex, but this is necessary to achieve scalability as the available number of transistors increases.

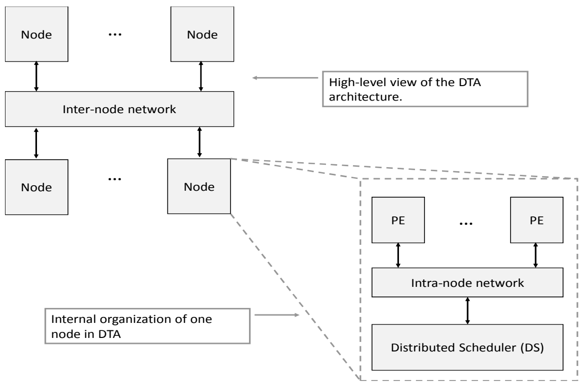

Node
...
Node
High-level view of the DTA architecture.
Inter-node network
Node
...
Node
PE
...
PE
Intra-node network
Internal organization of one node in DTA
Distributed Scheduler (DS)

Figure 1 – DTA architecture organization.

The logic for handling threads in DTA is distributed across PEs and nodes. Each PE contains one LSE (Local Scheduler Element) that manages local frames and forwards request for resources to the DSE (Distributed Scheduler Element). Each node contains one DSE that is responsible for distributing workload among processors in the node, and for forwarding it to other nodes when internal resources are depleted. For more details on both LSE and DSE see [2].

Figure 2 shows thread synchronization in DTA on code fragment from deblocking filter. The function for filtering MB has to filter all three color components for each edge in both directions. Therefore, in every pass it forks three threads for each color component (actually, it can be just for Y because Cb and Cr are compressed by sampling them at a lower rate to meet the storage and bandwidth limitations) and one thread that implements a barrier. In order to ensure that any thread won't start executing before all of its data is ready (so it can then run without blocking) a synchronization count (SC) has been associated to each of them. This synchronization count contains the number of input data that the thread needs in order to run. In our example, threads filter_mb_edgev and two instances of filter_mb_edgecv have to wait for just one input from filter_mb and since they are independent they can run in parallel. In reality, all of these threads consume more data but we presented only the most significant data to illustrate the concept. When data is stored for a thread, synchronization count is decremented and once it reaches zero that thread is ready to execute. Barrier thread has to wait signals from all three of these threads (SC=3) and then it can fork filter_mb thread for next pass.

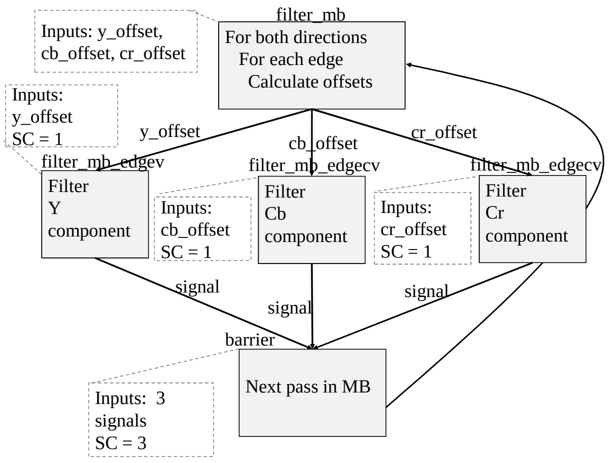


Figure 2 – Example of thread synchronization in DTA on a DF code fragment.

We have implemented two versions of deblocking filter for DTA architecture. One is sequential, where MBs are executed one by one and no other parallelism available is exploited. This code is for running on a single core only. The other code is parallel and it exploits all three levels of parallelism which are mentioned in section 2: independent MBs are processed in parallel, color components are processed in parallel and independent blocks of 4x4 pixels in vertical and horizontal passes are processed in parallel. We have to mention that, depending on input parameters, it is not always possible to exploit all these three levels of parallelism at the same time. Both versions of the code are handwritten. As a reference code, we have used a scalar implementation extracted from [12].

In DTA implementation, we didn't include deblocking filter parameter calculations, but just filtering itself. We suppose these parameters to be calculated by the previous steps and available as inputs of the program together with pixel color components.

4. Results

For our tests, we used first eight frames of Lake Wave video sequence. Frame resolution was 320x240 pixels – CIF resolution. For the DTA tests, we were using cycle accurate simulator with perfect memory model, written in C++. We extracted the data for the Cell processor from the work of Azevedo et al. [9].

Our first test was to measure the execution time reduction of each of the first eight frames of Lake Wave example by simply adding more processors in the system (all in a single node). Results are presented in Figure 3. We measured speedup using execution time on one processor as a baseline, for both sequential and parallel code. Execution time overhead of parallel code with respect to sequential is very low (about 3% on average). For this reason, speedup is very similar in both cases. As it is mentioned in section 2, the number of independent MBs in CIF resolution is at maximum 15 and little less than 9 on average, but it is increasing for higher resolutions. Therefore, from these results, we expect that even better speedups can be achieved for higher resolutions because more MBs are available in parallel. This means more threads with no dependencies among them. As stated earlier, not all three levels of parallelism are always available at the same time. That is why scalability is less than it could be expected theoretically.

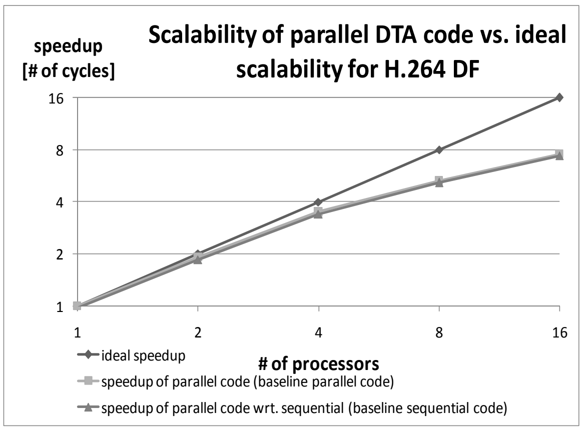
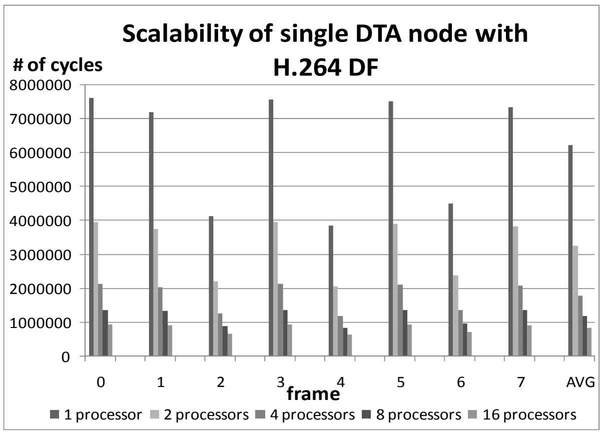


Figure 3 – Speedup of H.264 deblocking filter; DTA parallel code with a different number of processors in a single node vs. ideal case.

In Figure 4 we presented execution time for each frame for different number of processors in a single



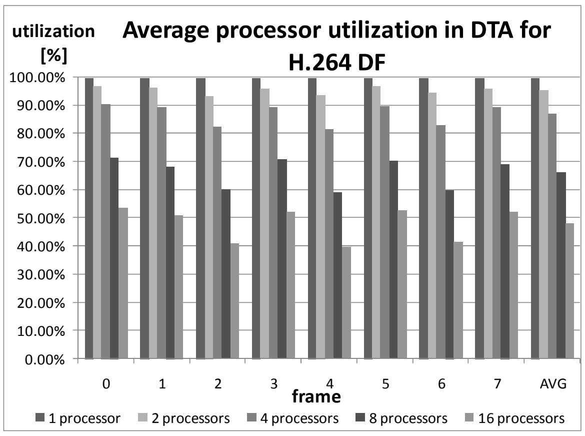


Figure 4 - Scalability of H.264 deblocking filter; speedup in execution time of first eight frames of Lake Wave video sequence by increasing the number of processors in a single node.

Figure 5 - Average processor utilization for first eight frames of Lake Wave video sequence by increasing the number of processors in a single node.

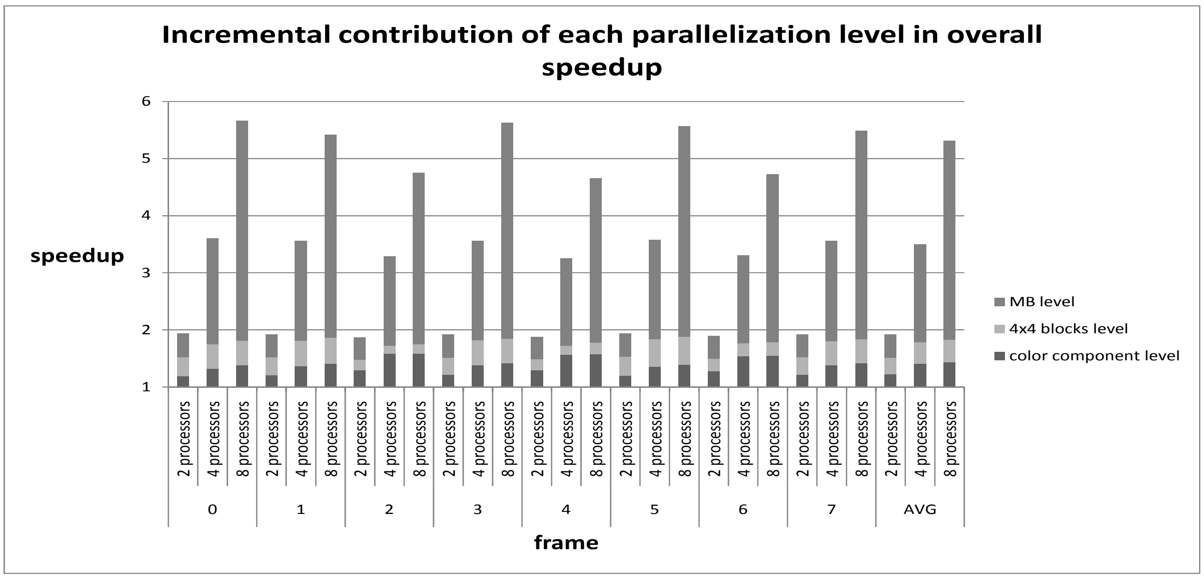


Figure 6 - Contribution of each parallelization in overall speedup; first is just with color component parallelization, then with 4x4 blocks parallelization included and at the end with all three types of parallelization together.

node. Execution time reduction is almost linear up to sixteen processors, but then it slows down because there is no more thread level parallelism available. Number of threads available remains the same even if we add more processors. However, threads are equally distributed among processors. That’s why we see lower average processor utilization in Figure 5 in the case of sixteen processor in a single node.

In Figure 6 we have presented contribution of each level of parallelization used in overall speedup. We measured these contributions incrementally. First we analyzed speedup just when processing color components in parallel. Then we added parallelism at 4x4 block level (MBs processed sequentially), and finally MB level parallelization was included. Baseline is the execution time on one processor (speedup equal to 1). It can be seen that for two processors contribution of each level is similar. However, for more processors in the system, overall speedup is dominated by MB level of parallelism and contribution of other levels doesn’t increase significantly. Available parallelism of color component level is limited by the fact that all three components are not processed in each pass (sub-sampling of Cb and Cr components). On the other hand, contribution of 4x4 block level parallelism is not at its theoretical maximum because this parallelism is dependent on input parameters (not in every case all blocks are processed) and also it introduces some overhead in order to be exploited. Overall conclusion is that MB level of parallelism is most significant, and with higher resolution it can increase even more, while other two are expected to remain at the same level.

Comparison with the real Cell is given just for the reference, as there are several differences in both cases. In DTA we assume for now a perfect memory model but we assume we could efficiently exploit double-buffering scheme as in the Cell [9]. On the other hand, we do not use software pipelining like in the Cell code. In the parallel version of the code for the Cell processor [9], sequential code is vectorized by hand utilizing SIMD capabilities of SPEs. As in DTA version, implementation doesn't include deblocking filter parameter calculations. Parallelization in Cell is based on SIMD ISA of SPEs and in DTA on adding more processors to exploit thread-level parallelism.

Our intention was not to compare performances of these architectures, but to show scaling possibilities of both of them. Figure 7 shows the results for two architectures. We presented the execution time of sequential and parallel versions of the code for both architectures (average for first eight frames of Lake Wave video sequence) and achieved speedup.

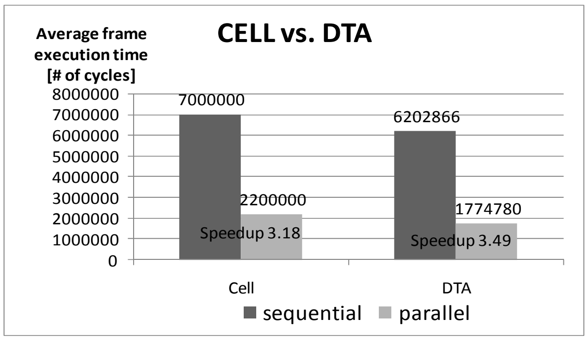


Figure 7 - Sequential and parallel execution of H.264 DF on IBM Cell and DTA; average of first eight frames of Lake Wave video sequence; in Cell parallel code is SIMD code able to execute 4 operations in parallel while in DTA there are 4 processors in a single node.

In the Cell, speedup is achieved by using SIMD capabilities of SPEs to execute four operations in parallel. In this way data level parallelism is exploited. Only one SPE is used for processing single frame. For the DTA architecture we showed execution time of sequential code running on a single processor and parallel code running on four processors in a single node. Reason for having sequential version result for DTA better than Cell is also because of a perfect memory model. Speedup achieved in DTA is 3.49 against 3.18 for Cell. In Cell, speedup is achieved by only exploiting ISA capabilities and in DTA by adding more processors. However, DTA uses very simple processors and it is fair to assume that it would be possible to put lot of them on a single chip. One additional thing to mention is that these two solutions exploit different parallelism that could eventually be combined to achieve even better results.

In the other tests, we were processing all eight frames together by distributing them among different nodes – system configurations from 1 to 8 nodes and from 1 to 16 processors in total. For distributing frames equally among nodes we used "ISA helped scheduling" [2]. Figure 8 shows speedup achieved for different system configurations. System configurations with the same number of processors in total, but distributed in more nodes (e.g. 2, 2 and 4, 1) has slightly worse performance due to the fact that the inter-node network has higher latency than intra-node network.

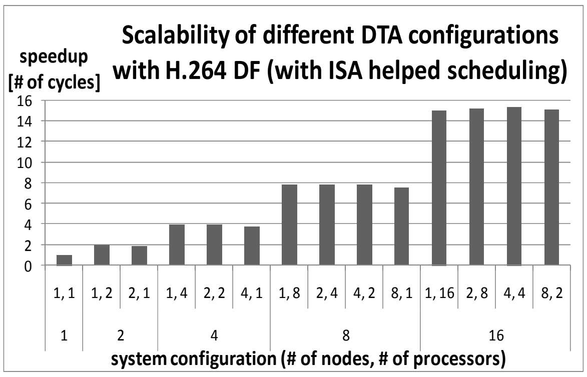


Figure 8 - Scalability of DTA with H.264 DF; speedup obtained by distribution of first eight frames of Lake Wave video sequence across nodes (1, 2, 4 and 8 nodes with different number of processors per node).

From Figure 9, we can see that average processor utilization in all of these cases is very high (more than 95% on average), which means that DTA architecture can efficiently exploit thread level parallelism in terms of non-blocking threads. In other words, if there is enough TLP in the program it can be efficiently exploited.

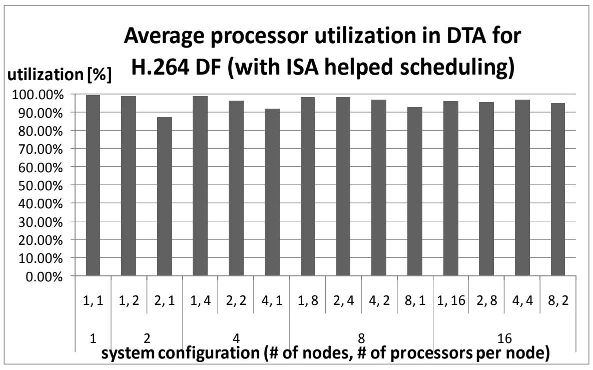


Figure 9 - Average processor utilization with H.264 DF in the case of distributing first eight frames of Lake Wave video sequence across nodes (1, 2, 4 and 8 nodes with different number of processors per node).

5. Conclusions

In this work we have presented parallelization possibilities of H.264 deblocking filter and its performance on DTA architecture. We have exploited three levels of thread level parallelism: macroblock level, color component level and parallel processing of portions of macroblocks.

We wrote parallel code for DTA by hand and executed it on a cycle accurate simulator. The results show that scalability of the architecture is very good. For up to sixteen processors it is almost linear, but after that the limits of available parallelism are reached. We have also shown a comparison with Cell processor with the goal to present scaling possibilities in both architectures. In Cell running SIMD version of the code on a single SPE speedup of 3.18 is achieved. In DTA architecture, by having four processors in the system we have achieved speedup of 3.49. In our case, the goal was to achieve scalability by simply adding more simple processing units. In this way, we have demonstrated that DTA architecture is suitable for accelerating portions of H.264 codec by parallel execution of deblocking filter.

As our future work, we plan to perform these tests on DTA architecture with more realistic memory system and with higher resolution inputs as well. Also, we want to investigate further possibilities for parallelizing other portions of H.264 codec.

Acknowledgement

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

References

[1] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC video coding standard," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 13, pp. 560-576, July 2003.

[2] R. Giorgi, Z. Popovic, and N. Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems," in *Proceedings of IEEE SBAC-PAD*, Gramado, Brasil, 2007, pp. 263-270.

[3] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," in *Proceedings of the 30th annual international symposium on Computer architecture* San Diego, California: ACM Press, pp. 422-433, 2003.

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.,* vol. 49, pp. 589-604, 2005.

[5] X. Zhou, E. Q. Li, and Y.-K. Chen, "Implementation of H.264 decoder on general-purpose processors with media instructions," in *Image and Video Communications and Processing 2003. Edited by Vasudev, Bhaskaran; Hsing, T. Russell; Tescher, Andrew G.; Ebrahimi, Touradj. Proceedings of the SPIE, Volume 5022, pp. 224-235 (2003).* 2003, pp. 224-235.

[6] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "A performance characterization of high definition digital video decoding using H.264/AVC," *Workload Characterization Symposium, 2005. Proceedings of the IEEE International,* pp. 24-33, 6-8 Oct. 2005.

[7] K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. A. S. H. Scale, "AltiVec extension to PowerPC accelerates media processing," *Micro, IEEE,* vol. 20, pp. 85-95, 2000.

[8] P. List, A. Joch, J. Lainema, G. Bjntegaard, and M. Karczewicz, "Adaptive deblocking filter," *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 13, pp. 614-619, July 2003.

[9] A. Azevedo, C. H. Meenderinck, B. H. H. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of Video Filtering on the Cell Processor," in *Proceeding in Prorisc Conference,* 2007.

[10] C. H. Meenderinck, A. Azevedo, M. Alvarez, B. H. H. Juurlink, and A. Ramirez, "Parallel Scalability of H.264," in *Proceedings of the first Workshop on Programmability Issues for Multi-Core Computers,* 2008.

[11] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation," *IEEE Transaction on Computers,* vol. 50, pp. 834-846, August 2001.

[12] "The FFmpeg Libavcoded," Available: http://ffmpeg.mplayerhq.hu/.