H.264 Parallel Optimization on Graphics Processors

Elias Baaklini*, Hassan Sbeity† and Smail Niar*

* University of Valenciennes, 59313, Valenciennes, Cedex 9, France
{elias.baaklini,smail.niar}@univ-valenciennes.fr
† Arab Open University, Beirut 2058 4518, Lebanon
hsbeity@aou.edu.lb

Abstract—**Multimedia applications are present in most mobile hand-held devices. The H.264 standard is currently dominating the video compression world. H.264 has high computational complexity requiring large amount of processing resources. Many techniques emerged that optimize H.264 using parallelization on multicore systems ranging from groups of pictures until the smallest block of pixels. We propose a parallelization technique based on rows of macroblocks with a light dependency detection algorithm that optimizes data parallelization and minimizes dependency synchronization stall time. The parallel H.264 implementation is tested on 2, 4, 8, and 16 cores processors using CIF and HD video resolutions benchmarks. The experimental results show that, in terms of execution time and parallel scalability, CIF video sequences peak at 4 cores with a speedup of 3.1 and HD video sequences peak at 8 cores with a speedup of 6.2. The H.264 parallel implementation is then tested on a graphics processor simulator of the Evergreen family of AMD GPUs reaching a speedup up to 12.1 times without communications overhead. Our results shall aid to find the best parallel configuration of the H.264 standard with the most suitable multicore platform to use in terms of time complexity and parallel efficiency.**

Keywords*—*Multimedia; H.264/AVC decoder; Video Compression; Optimization; Parallel Computing; Graphics Processors

I. Introduction

Multimedia hand-held devices are nowadays becoming more and more pervasive in many of modern world societies. Smart phones and tablet devices are equipped with high screen resolution and with relatively fast multicore embedded processors. DVD and blu-ray players, digital cameras, and LCD TVs support high resolutions like HD and Full-HD. However, few multimedia applications benefit from the computational potentials that multicore processors offer in these emerging powerful embedded devices. Furthermore, video coding standards like H.264/AVC [2] and HEVC [3] are adopting complex algorithms like context-adaptive binary arithmetic coding (CABAC) and variable length coding (CAVLC) in order to achieve better compression and thus lower transmission bitrates for high resolution video sequences. The additional complexity of these algorithms has a major impact by increasing execution time and energy consumption.

In our research, we intend to solve the problem of high complexity of the H.264 decoder using parallelization on multicore embedded processors and on graphical processors. Even with new cutting-edge processors, video resolutions are increasing rapidly, which require more processing time and consequently more energy consumption. Many solutions based on parallel execution exist ranging from macroblocks (fine-grain) till groups of pictures (coarse-grain) parallel decoding. Macroblock parallel decoding is highly scalable since many macroblocks can be processed in parallel. However, dependencies and huge overheads are created as a result of communication and synchronization between macroblocks. Parallel decoding of groups of pictures require large memories for high definition video sequences. In addition, they have a lower scalability than macroblock decoding because of the limited number of groups of frames that can be decoded in parallel. Our solution is to decode macroblock rows in parallel. This level of parallel execution is considered between the coarse-grain and the fine-grain parallelization approaches. It also offers a balance between large overheads and high scalability of previous solutions.

Our main contribution in this paper is the design of a new approach for the parallelization of the macroblock rows of the H.264 decoder with an algorithm that detects dependencies on-the-fly based on isolating intra-prediction macroblocks (I-MBs). Experiments are conducted using simulations on 2, 4, 8, and 16 cores processors. We further experiment our parallel implementation on a graphical processor simulator of the Evergreen AMD GPU. We compare CPU and GPU experimental results. Our results define the best multicore processor with the highest speedup and the best parallel efficiency. For CIF resolutions, video sequences benchmarks reach their maximum throughput using 4 cores with a speedup of 3.1. For HD video sequences, 8 cores processors offer the best time and energy efficiency combined with a speedup of 6.2. On a GPU with 16 parallel computational units, the speedup reaches 12.1 for HD resolutions and 7.4 for CIF resolutions.

In our H.264 parallel implementation, the motion compensation (MC) stage for each row of inter-prediction macroblocks (P-MB) is executed in parallel on different cores. We experiment the parallel version using low and high definition resolutions, CIF and HD respectively, on multicore processors. Macroblock dependencies in the same picture slice are avoided by decoding intra-prediction macroblocks (I-MBs) when all other macroblocks are decoded. Overheads emerged as a result of shared memory communications and synchronization between cores. We simulated the parallel execution on multicore processors and graphical processors using a multicore simulator, Multi2Sim [8]. We further investigate the scalability of the multiple cores implementation, which shows the existence of a virtual threshold depending on the resolution when large numbers of cores are used.

The remainder of the paper is organized as follows. In Section 2, we present the related work concerning H.264 parallel optimizations. In Section 3, we describe our approach for parallel execution of macroblock rows of the H.264 decoder. In Section 4, we present the experimental results for execution time on CPUs and GPUs using a simulator for multicore processors. Final conclusion and future work are given in Section 5.

II. Related Works

Ever since the H.264/AVC standard [2] was published in 2003, researchers started to solve the high complexity issue of the new standard mainly using parallelism. Several modifications were suggested for the H.264 encoders and decoders in order to improve the performance in terms of execution time and memory usage. Parallel decoding techniques of H.264 exist from the highest level, which is the group of frames or pictures (GOP), the coarse-grain level, till the lowest level, which is the block inside a macroblock, the fine-grain level. Kannangara [12] reduced the complexity of the H.264 decoder (19-65%) by predicting the SKIP macroblocks using an estimation based on a Lagrangian rate-distortion cost function. Gurhanli [14] suggested a parallel approach by decoding independent groups of frames on different cores. The speedup is conditioned with the modification of the encoder in order to omit the start-code scanner process. Any modification to the encoder will require a long process for modifying the H.264 specification in order to be compliant with the standard. The exclusion of previously encoded video sequences is also an effect for modifying the H.264 encoder. Nishihara [18] proposed a load balancing mechanism among cores where partitions sizes are adjusted during runtime. He also reduced the memory access contention based on execution time prediction. Among frame-level and MB-level parallelization, the 3D-wave technique proposed by Azevedo [15] decodes independent MBs in parallel on different cores. A good scalability is proved for HD resolutions where macroblocks are scanned in zigzag mode and decode independent macroblocks in parallel. Chong [16] added a pre-parsing stage in order to resolve control dependencies for MB-level parallelization. Van Der Tol [20] mapped video sequences data over multiple processors providing better performance over functional parallelization. He groups macroblocks in a way that minimal dependency between cores is required. Horowitz [17] compared different H.264 implementations including FFmpeg [4] and the H.264 reference software JM [1]. He also analyzed the complexity of the H.264 decoder subsystems. Sihn [19] proposed a multicore pipeline for the deblocking filter based on the group of pictures data level partitioning. He also suggested software memory throttling and fair load balancing techniques in order to improve multicore processors performance when several cores are used. In our research we optimize the H.264 decoder knowing that our approach can be also applied to the H.264 encoder. We focus on improving the efficiency of the H.264 decoder using multicore processors. We decode rows of macroblock in parallel where rows are mapped to a number of cores. Dependencies between macroblocks are avoided by decoding intra-prediction macroblocks sequentially at the end of the decoding stage. We map our implementation on 2, 4, 8, and 16 cores. Speedup of the parallel implementation is calculated using simulated execution time. We further implement an OpenCL [5] version of our parallel H.264 implementation. Simulation experiments on graphics processors are conducted using a CPU-GPU simulation Multi2Sim [8].

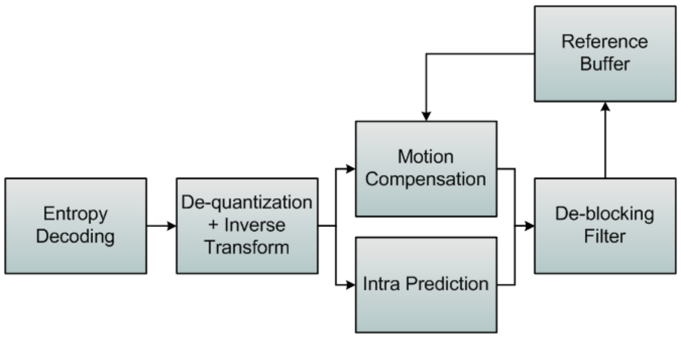


Figure 1. H.264 decoding process

In the following section, we describe in detail our parallel implementation of the H.264 decoder. In addition, we describe our environment configuration for the execution simulations on normal processors and graphics processors.

III. H.264 Parallel Implementation

In this Section, we describe our parallel implementation of the H.264 video decoder. We start with a brief overview of the decoder, then we explain how we parallelize the decoder, and finally we compare our approach to other similar parallel implementations.

A. Parallel Execution and Synchronization

Parallel execution is considered as a major potential solution for complex applications where sequential execution bounds the performance of these applications. Most processors that are currently available in the market have multiple cores and support many threads. Applications with low execution efficiency may benefit from a high potential speedup when data or functional parallelization is applicable. Even optimized implementations can still take advantage from parallelization techniques. In our research, we choose the H.264/AVC video decoder as our multimedia application benchmark for which we provide a parallel implementation using our approach. We further gather execution statistics and we compare results to other relatively similar implementations.

B. H.264 Standard

The Moving Picture Experts Group (MPEG) and the Video Coding Experts Group (VCEG) developed jointly in 2003 the ”Advanced Video Coding” (AVC) standard published as ITU-T Recommendation H.264 and as part 10 of MPEG-4. Since the first commercial implementations, several multimedia device manufacturers adopted the new video codec. About 7 years after the first release of the final draft of the standard, H.264 is the mostly used video compression standard in most multimedia devices according the PCWorld.com [6]. Cameras, smart phones, PDAs, CCTV recorders, blu-ray disc players and many other devices use H.264 for encoding and decoding videos. H.264 achieves better compression and higher quality at the expense of more complex algorithms. Thus more computation

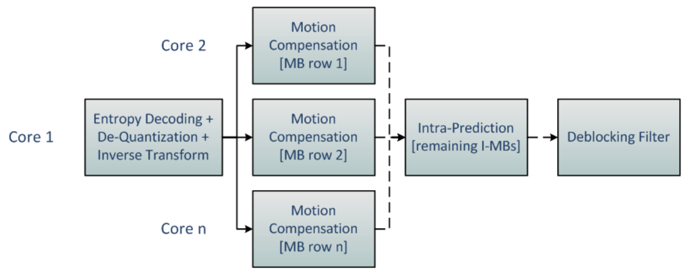

Core 1
Core 2
Core n
Entropy Decoding + De-Quantization + Inverse Transform
Motion Compensation [MB row 1]
Motion Compensation [MB row 2]
Motion Compensation [MB row n]
Intra-Prediction [remaining I-MBs]
Deblocking Filter

Figure 2. Decoding macroblock rows in parallel on n cores

resources are exploited and more energy is consumed in order to increase compression ratio of video files.

C. H.264 Decomposition

The H.264 decoder can be divided into five main functional parts: Entropy Decoder (ED), De-Quantization and Inverse Transform (IQT), Motion Compensation (MC) and Intra-Prediction (IP), and Deblocking Filter (DF). The H.264 decoder stages are illustrated in Figure 1. A slice of a picture is partitioned into blocks of 16 x 16 pixels called Macroblock (MB). The number of horizontal macroblocs and vertical macroblocks varies with the resolution of the frame picture that is being decoded. Entropy decoding is performed for all bits in a slice of a frame. Motion compensation or intra-prediction is applied for every macroblock of size 16 x 16 pixels. A macroblock can be also divided into sub-blocks of 16 x 8, 8 x 8, 8 x 4, and 4 x 4 pixels. The encoder chooses the sub-blocks sizes depending on the image complexity of the video sequences being decoded. The motion compensation stage uses a reference buffer in order to calculate the values of macroblocks in the current frame. The reference buffer contains a list of previously decoded frames. Macroblocks that are inter-predicted and motion compensated from previously decoded frames are either of type P or B (P-MBs and B-MBs). Macroblocks that depend on macroblocks in the current frame (called I-MBs) are intra-predicted. Deblocking filter is executed at the end of the decoding process in order to reduce the edging effect between macroblock borders.

D. Parallel Execution

The H.264 reference software, JM [1], is an open source implementation used as a reference implementation for the H.264 standards. In our research, we modified the JM [1] source code of the H.264 decoder in order to decode rows of macroblocks in parallel using the PThread library in C programming language. As shown in Figure 2, each core handles the motion compensation stage for macroblocks in a group of rows. Motion compensation and intra-prediction phases should be completed before applying the DF phase. Data parallelization is applied to the motion compensation phase of different macroblocks. The maximum numbers of parallel data execution is equal to the number of macroblock rows. One of the available cores is needed to coordinate the execution of the parallel decoding process on different cores. The coordinating core may be one of the cores that are used for parallel execution since parallel cores are only used for part of the decoding process. The level of parallel decoding of

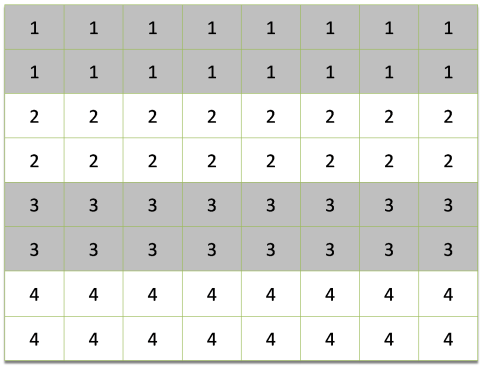
```
input: list of macroblocks MBlist
       Number of cores cores

list of I-MBs DepMBlist in a slice

// main loop where each iteration is for 1 core
for each subMBlist of MBlist do // depending on cores
   for each element currMB in subMBlist do
        if currMB.TYPE == I16MB
              or currMB.TYPE == I8MB
              or currMB.TYPE == I4MB
        then
              DepMBlist.add(currMB);
              continue;
        else
              decode(currMB);
        endif
   endfor

   for each element currMB in DepMBlist do
        decode(currMB);
   endfor
endfor
```

Figure 3. Decoding rows of macroblock with intra-prediction dependency check algorithm

macroblock rows may be considered between coarse-grain and fine-grain approaches. High level approaches process multiple slices or frames in parallel. Low-level approaches decode macroblocks or blocks inside a macroblock in parallel. This balance between both approaches is also reflected between synchronization overheads and memory requirements. Coarse-grain methods need high memory usage in order to decode multiple frames in parallel. Fine-grain methods cause an enormous synchronization overhead affecting deeply the speedup. Our approach is aimed to benefit from the balance between both advantages and disadvantages. Macroblock rows require less memory than a frame and more than one macroblock. The number of rows is much less than the total number of macroblocks. For example, in HD resolution (1280 x 720), each frame has 3600 MBs, 80 horizontal MBs and 45 vertical MBs. Thus, the number of macroblocks rows is less by a factor of 80 than the total number of macroblocks. As a result, the overhead for synchronization and communications between cores is also reduced by a factor of 80.

E. Dependencies between Macroblocks

In H.264, there are 3 types of macroblocks: I, P, and SKIP. I-MBs depend on other macroblocks in the same slice of a frame. P-MBs depend on macroblocks from previously decoded frames. SKIP-MB uses the same macroblock from a reference frame without transmitting the motion vector information. I-MBs require dependent macroblocks, which are in the same slice, to be previously decoded. So a dependency identification procedure is needed in order to satisfy I-MB dependencies. In order to overcome these dependencies, we start by decoding all P-MBs and SKIP-MBs rows in parallel. When this operation is completed, the remaining macroblocks, which are I-MBs in the current slice, are decoded sequentially. With this simple ordering, dependencies between macroblocks in the same slice are satisfied. The average number of I-MBs in P-Frames and B-Frames is 2.5% for CIF resolution and 4% for HD resolution. A video sequence always starts with an I-Frame (IDR), which is composed completely of I-MBs. This



1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
4	4	4	4	4	4	4	4

Figure 4. Speedup of the motion compensation and intra-prediction stages on multicore processors

type of frames is available typically every 150 to 200 frames (3 to 8 seconds depending on frame rate) in a video sequences in order to overcome communication problems when some frame data are lost during communication transmission. We can increase or decrease the frequency of IDR frames in the encoder configuration. However, a high frequency of IDR frames, for example one I-frame every 30 or 50 frames, will significantly decrease the compression ratio of the decoder. The number of I-MBs in a P-Frame or a B-Frame depends on the complexity of the image and on the objects in the image and their rate of movements in the video sequences. P-Frames and B-Frames are mostly composed of P-MBs and SKIP-MBs with a small number of I-MBs. So the number of I-MBs does not significantly affect the overall speedup for the parallel decoding of MBs. Figure 3 shows the pseudocode for the macroblock dependency check algorithm in addition to the iteration over macroblock rows in a slice of a frame and the assignment of macroblock rows to different threads or cores of a processor. The list of all macroblocks and the number of cores are given as input data. A main loop iterates over groups of macroblocks assigned for each core. This loop is mapped onto the assigned cores in order to be executed in parallel. An inner loop checks every macroblock. I-MBs are added to an empty list. The remaining macroblocks are decoded. After the main loop, a second loop iterates over all I-MBs that are in the new list and decodes all the macroblocks in the list.

F. *Macroblocks Partitioning*

In a frame slice, while iterating over macroblocks, we skip intra-prediction macroblocks (I-MBs) and we decode P-MBs and SKIP-MBs in parallel on multiple cores as described above in the algorithm in Figure 3. Depending on the number of available cores, we group rows of macroblocks in order to be decoded in parallel. The slice is divided by the number of cores horizontally. In [7], 6 parallel representations are experimented in terms of stall time and core usage. Among the presented data partitioning approaches, our partition is similar to the slice-parallel splitting approach that is described in [7]. As shown by the author, this approach has a high stall time overhead. This stall time is caused by synchronization procedures in order to satisfy macroblock dependencies. However, with our approach for avoiding dependencies between macroblocks, the stall time overhead does not apply. We chose this method

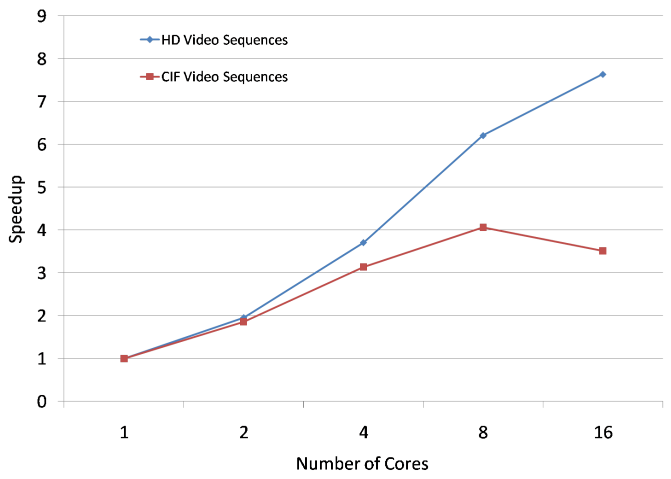


Figure 5. Speedup of the motion compensation and intra-prediction stages on multicore processors

TABLE I. Speedup of video sequences on multicore processors

Resolution	MB Rows	2	4	8	16
HD (1280 x 720)	45	1.959	3.710	6.207	7.636
CIF (352 x 288)	18	1.861	3.142	4.065	3.517

because of data locality and because of minimal data transfer initiation overhead. For example, in order to execute a slice of 80 rows of macroblocks on 4 cores processor, each core decode a chunk of 20 rows of macroblocks. Using this partition method, data is only transferred 4 times to the cores, which is the minimal number of transfers because it is equal to the number of available cores. In Figure 4, we show en example of a frame of size 64 x 64 pixels, 8 x 8 MBs, mapped onto 4 cores. The numbers inside the squares are the numbers of cores. Macroblocks in Figure 4 are assumed to be all P-MBs or B-MBs. I-MBs are not displayed for illustration purposes.

G. *GPU Parallelization*

The H.264 decoder parallel implementation is further modified to execute motion compensation process of P-MBs and B-MBs on graphics processors (GPUs). Part of the code is modified in order to comply with OpenCL [5] language, which is a unified framework for defining and controlling a GPU. Kernels, functions in C language, written in OpenCL are executed on a graphics device. Parallel execution of groups of macroblock rows are processed by work-groups. Slice data is first transferred to the graphics device and transferred back to the memory of the processor in order to complete the decoding process. Parallel execution of the motion compensation stage is performed as illustrated in Figure 2 where work-groups are considered as cores. Experimental results and comparisons with processor execution are discussed in the following section.

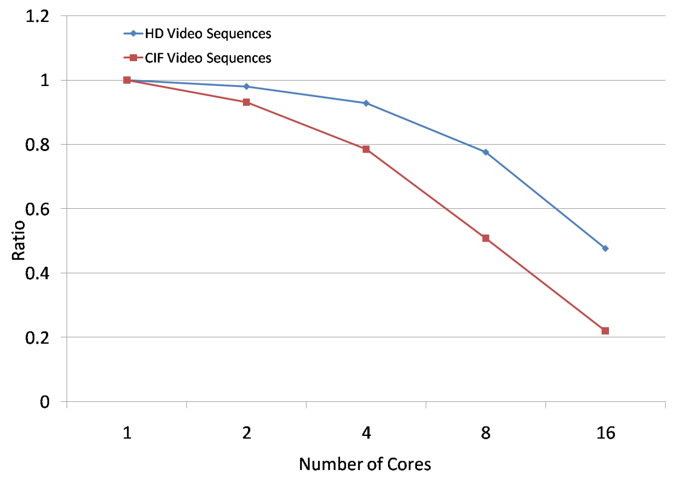


Figure 6. Speedup to number of cores ratio

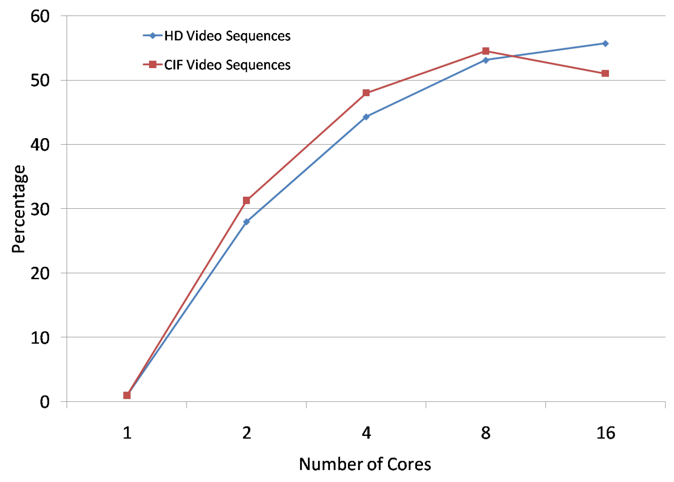


Figure 7. Percentage gain for the complete decoding process on multicore processors

IV. Experimental Results

In this Section, we test our H.264 parallel implementation using multicore x86 processors and graphics processors. We gather simulation statistics and we compare our results with other results for parallel H.264 implementations.

A. Simulations

Our H.264 parallel implementation described in Section 3 is executed by Multi2Sim [8], a cycle-accurate simulator for multicore x86 and graphics processors. Cache and memory configurations comply with common x86 processors that are available nowadays in many Intel [11] or AMD [9] processor chips. Each core has a private L1 cache of 512 KB and All other cores have a shared L2 cache of 2 MB. We simulate the execution of our parallel H.264 decoder using 2, 4, 8, and 16 cores processors. We perform simulation experiments of the H.264 OpenCL version on the AMD Evergreen GPU family with the configurations of the AMD Radeon 5870 GPU [10]. We gather statistics using 3 video sequences with CIF resolution (bus, waterfall, and flowers) and 3 video sequences with HD resolution (Intotree, Parkrun, and Shields). Simulation is performed for the H.264 decoding process of 60 frames for each video sequence.

B. Results

Execution times with different number of cores using CIF and HD resolutions for the motion compensation stage are listed in Table I. The number of parallel rows of macroblocks increases with the video resolution. Thus HD resolution scales better than CIF resolution with the number of core. Experiments are conducted using simulations on 2, 4, 8, and 16 cores processors. Speedup results for the motion compensation stage are illustrated in Figure 5. For CIF resolutions, the maximum speedup of 4 is attained using 8 cores. With 16 cores, the speedup decreases to 3.5 due to large data communication overhead. For HD video sequences, a 7.6 speedup is reached with 16 cores processor. These optimization speedups are not efficient when compared to the number of cores used. Figure 6 shows that the ratio between the number of cores and the speedups is very high when using 16 cores. The best efficiency ratio is 4 cores with a speedup of 3.1 for CIF resolution and 8 cores with a speedup of 6.2 for HD resolution. The ratio of the speedup to the number of cores using 4 cores for CIF and 8 cores for HD is around 0.8. Doubling the number of cores drops the ratio to 0.5, which cannot be considered as efficient as we expect when running a parallel application on a multicore processor. In [13], the highest speedup is 5 on 8 cores and 8.1 on 16 cores. Our results have a better ratio, for less than 16 cores, related to the number of cores. For 16 cores and above, the results in [13] are better. However, decoder implementation, processor configurations and video resolutions vary between both approaches. Thus, exact comparisons are not applicable. The overall performance gain for all stages of the H.264 parallel decoder is illustrated in Figure 7. CIF resolutions reach 48% increase in performance using 4 cores and HD resolutions attain 53.1% using 8 cores.

C. Parallel Execution on Graphics Processor

We experiment our parallel implementation on a graphical processor simulator of the Evergreen AMD GPU. Figure 8 shows the speedups attained with the GPU devices. HD resolutions have a speedup of 12.1 and CIF resolutions a speedup of 7.4. These results exclude the data transfer time between the main processor and the graphics processor. This overhead limits the usability of the GPUs when the gain is low. In our case, the ratio of the speedup to the number of work-groups is around 0.75. Speedup simulation results for CIF and HD resolutions decoding on GPU are displayed in Table II. Graphics processor have high potential of parallel optimization. The number of work-groups and work-items is increasing significantly in new devices. Hundreds of work-groups and thousands of work-items can have a huge impact on applications with with high parallel data processing.

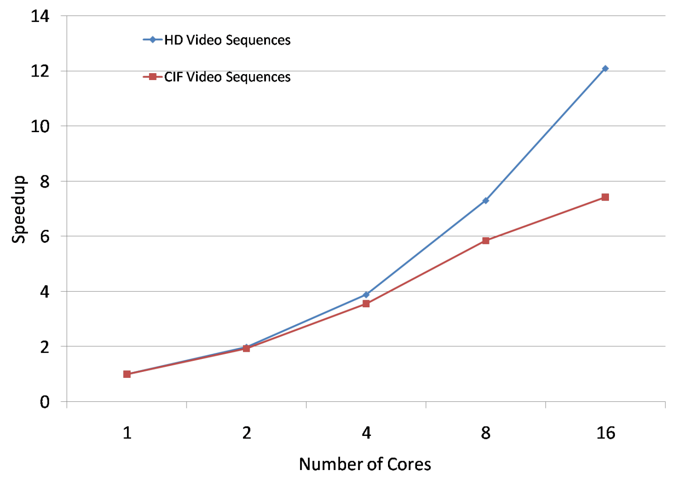


Figure 8. Speedup of H.264 parallel execution on Evergreen GPU

TABLE II. Speedup of video sequences on graphics processors

Resolution	MB Rows	2	4	8	16
HD (1280 x 720)	45	1.983	3.884	7.301	12.095
CIF (352 x 288)	18	1.928	3.560	5.839	7.417

V. Conclusion and Future Works

We have introduced a novel parallel technique for H.264 video decoder. Our approach decodes in parallel macroblock rows of the H.264 decoder with an algorithm that detects dependencies on-the-fly based on isolating intra-prediction macroblocks (I-MBs). Experiments using CIF and HD video sequences show that every resolution has a virtual threshold for the speedup when increasing the number of cores. This limit is due to the increase of data transfer between cores. The best speedup with the highest ratio to the number of cores is 3.1 for CIF resolutions using 4 cores and 6.2 for HD resolutions using 8 cores. A speedup of 12.1 is attained the H.264 parallel implementation is executed on a graphics processor. Additional research and experiments need to be conducted on the OpenCL implementation for GPUs. We plan to test our implementation on real boards and gather more statistics like memory usage and power consumption in addition to execution time and optimization efficiency in general.

References

[1] K. Suhring. H.264 reference software. http://bs.hhi.de/ suehring/tml/.

[2] AISO/IEC. International standard. Part 10: Advanced video coding, 14496-10, 2003.

[3] JCT-VC. High efficiency video coding (HEVC) text specification draft 8. 10th Meeting: Stockholm, SE, 1120 July 2012.

[4] FFmpeg project. http://www.ffmpeg.org/.

[5] OpenCL: The Open Standard for Parallel Programming of Heterogeneous Systems. http://www.khronos.org/opencl.

[6] IDG Consumer & SMB. PCworld Magazine. http://www.pcworld.com/.

[7] F. Seitner, M. Bleyer, M. Gelautz, R. Beuschel. Evaluation of data-parallel H.264 decoding approaches for strongly resource-restricted architectures. Multimedia Tools and Applications, 1(2010), S. 1 - 27.

[8] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. Proc. of the 21st International Conference on Parallel Architectures and Compilation Techniques, Sep., 2012.

[9] AMD Opteron Processor Family. http://www.amd.com/.

[10] AMD Evergreen Family Instruction Set Arch. (vl.Od). http://developer.amd.com/sdks/amdappsdk/documentation/.

[11] Intel Core Processor Family. http://www.intel.com/.

[12] C. S. Kannangara and I. E. G. Richardson and M. Bystrom and J. Solera and Y. Zhao and A. Maclennan Complexity reduction of H.264 using Lagrange Optimization Methods. IEE VIE 2005, Glasgow, UK, 2005.

[13] M. A. Mesa, A. Ramirez, A. Azevedo, C. Meenderinck, B. Juurlink, and M. Valero. Scalability of Macroblock-level Parallelism for H.264 Decoding. Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems, pages 236–243, ICPADS, 2009.

[14] A. Gurhanli and S. Hung. Coarse grain parallelization of h.264 video decoder and memory bottleneck in multi-core architectures. International Journal of Computer Theory and Engineering vol. 3, no. 3, pages 375–381, 2011.

[15] A. Azevedo, C. Meenderinck, B. Juurlink, A. Terechko, J. Hoogerbrugge, M. Alvarez, and A. Ramirez. Parallel h.264 decoding on an embedded multicore processor. HiPEAC, pages 404–418, 2009.

[16] J. Chong, N. Satish, B. Catanzaro, K. Ravindran, and K. Keutzer. Efficient parallelization of h.264 decoding with macro block level scheduling. ICME, pages 1874–1877, 2007.

[17] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro. H.264/avc baseline profile decoder complexity analysis. IEEE Trans. Circuits Syst. Video Techn., 13(7):704–716, 2003.

[18] K. Nishihara, A. Hatabu, and T. Moriyoshi. Parallelization of h.264 video decoder for embedded multicore processor. ICME, pages 329–332, 2008.

[19] K. Sihn, H. Baik, J. Kim, S. Bae, and H. Song. Novel approaches to parallel h.264 decoder on symmetric multicore systems. Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 09, pages 2017–2020, Washington, DC, USA, 2009. IEEE Computer Society.

[20] E. Van Der Tol, E. Jaspers, and R. Gelderblom. Mapping of h.264 decoding on a multiprocessor architecture. Image and Video Communications and Processing, pages 707–718, 2003.