International Journal of Control and Automation
Vol.7, No.6 (2014), pp.379-388
http://dx.doi.org/10.14257/ijca.2014.7.6.35

A Parallelized Implementation for H.264 Real-time Encoding Scheme

Young Chun Kwon[1,2], Chanho Park[3], Daeseok Oh[3],
WooSuk Jang[4] and Nakhoon Baek[1]
[1]*School of Computer Sci. and Eng., Kyungpook National University, Daegu 702-701, Korea*
[2]*Samsung Electronics, Suwon, Korea*
[3]*Kumoh National Institute of Technology, Gumi 730-701, Korea*
[4]*Yeungnam University, Gyeongsan 712-749, Korea*

kown100@nate.com, cksgh303@gmail.com, rar555@nate.com, passions707@naver.com, oceancru@gmail.com

Abstract

In this paper, a high-speed video stream encoder for the H.264 digital video codec standard specification is accelerated with nowadays parallel processing architectures. Based on the parallel processing techniques with GPU's, we used an OpenCL-based GPU kernel programs, and finally achieved a high-level CPU-GPU interoperability. In its design, our system makes the CPU perform all input/output operations and overall stream control, while GPU does the core encoding operations. Our final result shows remarkable speed-up in comparison with the previous implementations. All the details of our implementation are presented in this paper.

Keywords: *Video codec, H.264, OpenCL, parallel processing*

1. Introduction

In these days, H.264 codecs are the most widely used digital video codecs [1, 2, 3]. The overall encoder processing requires huge amount of computations, to guarantee the final video quality. Although most previous H.264 encoder implementations are focused on their CPU-based accelerations, there are a few results on the parallelized H.264 encoders, recently.

In this paper, we represent a new acceleration method for the H.264 decoders, based on the OpenCL (Open Computing Language). OpenCL [4] is currently one of the most widely used de facto standards in the field of general-purpose GPU (GPGPU) computing. Most previous researchers are concentrated on the acceleration of CPU computing, or introducing intuitive parallel processing architectures. In contrast, we focused on the overall parallelization of the encoding operations, and finally achieved remarkable speed-ups.

In Section 2, we present previous CPU-based and GPGPU-based encoding schemes for the H.264 encoder. Our overall parallelized architecture and details of implementation algorithms are followed in Section 3. We show the actual implementation results in Section 4. Section 5 shows our conclusion and future works.

[1] Corresponding author: Nakhoon Baek, oceancru@gmail.com

[2] This investigation was financially supported by Semiconductor Industry Collaborative Project between Kyungpook National University and Samsung Electronics Co. Ltd.

ISSN: 2005-4297 IJCA
Copyright ⓒ 2014 SERSC

2. Previous Works

GPGPU-based methods have been widely used for not only encoder processing but also various areas including image processing, big-scale data processing, and others. Specifically for the encoder processing, CUDA [5] and OpenCL are used for various video encoding methods to enhance their internal algorithms. In the case of H.264, it shows relatively good encoding performance, and there are CUDA-based acceleration examples [6] and also OpenCL-based acceleration examples [7].

In this paper, we are focusing on the OpenCL standard. In the case of CUDA, it was released from a single manufacturer, NVIDIA, and thus, it cannot be easily used for the general devices from other hardware vendors.

There is a previous work [8] for the H.264 encoding pipeline implementation, with OpenCL. Our work is based on their implementation. We analyzed their overall architecture and located the key operations to be parallelized. Using OpenCL, we accelerated those key operations to finally achieve more optimized performance.

3. Design and Implementation

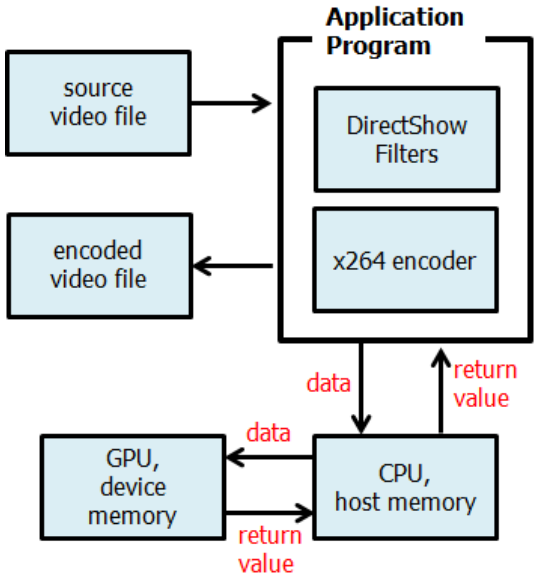


Figure 1. Overall architecture of our parallelized H.264 encoder

Figure 1 shows the overall architecture of our proposed algorithms. To concentrate on the encoding operations, we used the input/output features of the existing DirectShow filters [9] for the file input/output operations. When the target video files are read from the disk images, the DirectShow filters construct the raw image data in the host memory. Then, these image data are sent to the GPU, and the OpenCL-based GPU kernel program does the core encoding operations. The finally encoded results are copied back the host memory, and the DirectShow filter converts them to the final encoded video files on the disk.

In this paper, we will concentrate on the *x264 codec*, which is currently one of the most powerful H.264 codecs, even with open sources. This x264 codec is a freeware library for H.264/AVC video stream encoding, but without any decoder. Although x264 library is open sourced freeware, it shows relatively good performance, and widely used for many

 Copyright ⓒ 2014 SERSC

applications. Our focus is accelerating this H.264 encoding library through applying GPU operations. Thus, we first analyze the internal structure and data flow of x264 library, and then, select some specific portions to be accelerated with GPU.

Although the DCT operations and quantization are different in their required operation, both of them perform repeated processing for each image block. More precisely, it starts from the *intra image predictions* for *I*-slices of the H.264 video stream. In x.264 standard specification, most operations, including DCT and quantization, are designed to be applied to an image block. Thus, in its implementation, the repeating loop structures are bounded to at most 16 times. Due to this fact, we tried to convert these loop structures to OpenCL parallel processing units.

At the early stages of our parallelized program design, we analyzed that the overall operations are concentrated on the per-screen operations. Thus, we first tried to optimize these per-screen operations. However, our optimization immediately causes rapid increase of data transfer between the GPU and host memory. This kind of bottle-neck problem limits the overall performance of our architecture. Thus, we changed our optimization strategy to optimize another processing step of look-ahead operations. The look-ahead operations examine the future frames in the video stream, and decide the frame types and slice types of them in advance, according to the H.264 standard specification.

The whole look-ahead processing is implemented as a single thread, and this thread uses the OpenCL-based GPU kernel to determine each slice type. The OpenCL kernel internally performs 4 steps: *down-scale*, *intra-prediction*, *motion-search*, and *mode select*. At the down-scale stage, the H.264 macro blocks are converted into lower resolution images. And then, the intra-prediction stage calculates the intra-cost estimation values. At the motion search stage, the motion vectors are obtained from the scaled-down images and intra-cost values. At the mode select stage, each slice types are selected from the cost values from the intermediate results.

Figure 2 represents the overall structure of the look-ahead algorithm. The *slicetype_decide* function performs analysis of each slide and finally determines the slice type. That function executes its sub-function of *slicetype_frame_cost* for each macro block. We have implemented the sub-function *slicetype_frame_cost* in OpenCL, to achieve GPU accelerations. Each OpenCL command from the host program is entered to the command queue and moved to the device. The OpenCL kernel runs with the processing element unit, with respect to the pre-specified dimensions and sizes. The frame images are referenced with the OpenCL image object, and the resulting images are stored to the host memory buffer, with OpenCL memory object.

Internally, the OpenCL kernel function performs downscale, intra prediction, motion search, and mode select stages in that order, as shown in Figure 2. At the downscale stage, the input macro block is processed to make a low resolution image from the original image. Then, the intra prediction stage analyzes that low resolution image to calculate the corresponding *intra_cost*. Based on the low resolution image and its corresponding *intra_cost*, the mode search stage extracts the motion vectors. The final mode select function calculates the final cost, which represent the best suitable slice type.

Copyright ⓒ 2014 SERSC

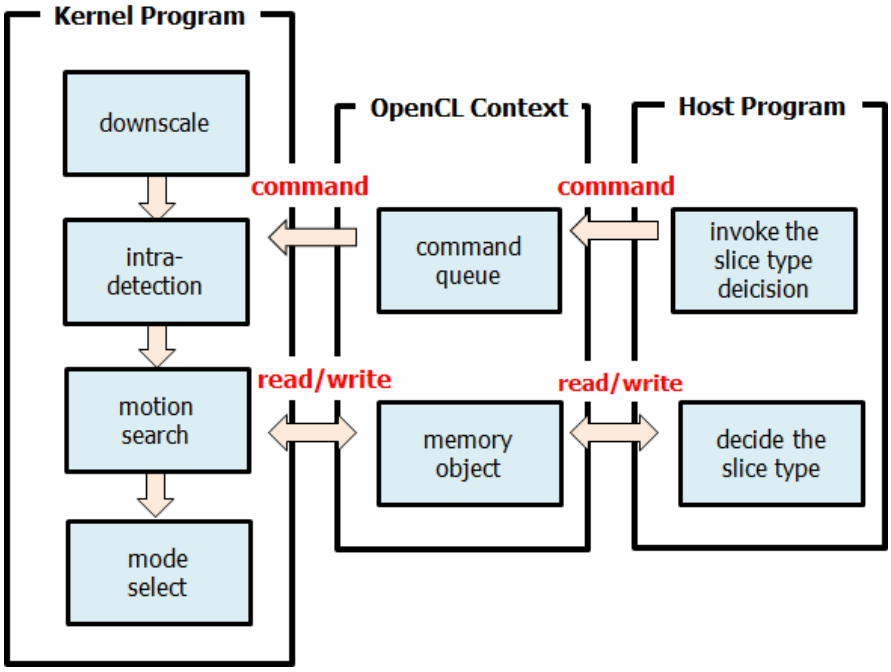


Figure 2. Our look-ahead algorithm implementation

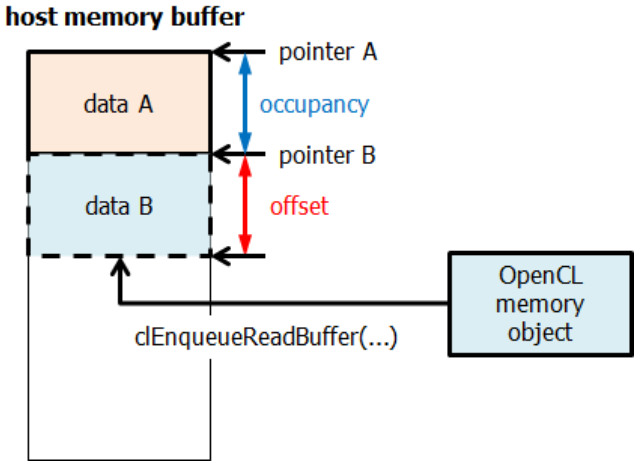


Figure 3. Buffer pool management

For more optimized OpenCL memory management, the low resolution image and motion vectors are stored in the global memory area. Intermediate results are stored in the private memory area. The final result of OpenCL kernel program needs to be transferred to the host program, with *clEnqueueReadBuffer* function.

Notice that the host memory and the device memory areas are isolated, and thus, each memory area needs its own pointer management. In our system, we use a base pointer to the buffer pool and an offset value from that base location, as shown in Figure 3. For more efficiency, the contents of memory buffer are transferred with memory copy functions, only when the buffer becomes full. After memory copy operation, the buffer and its related pointers are reset.

Copyright ⓒ 2014 SERSC

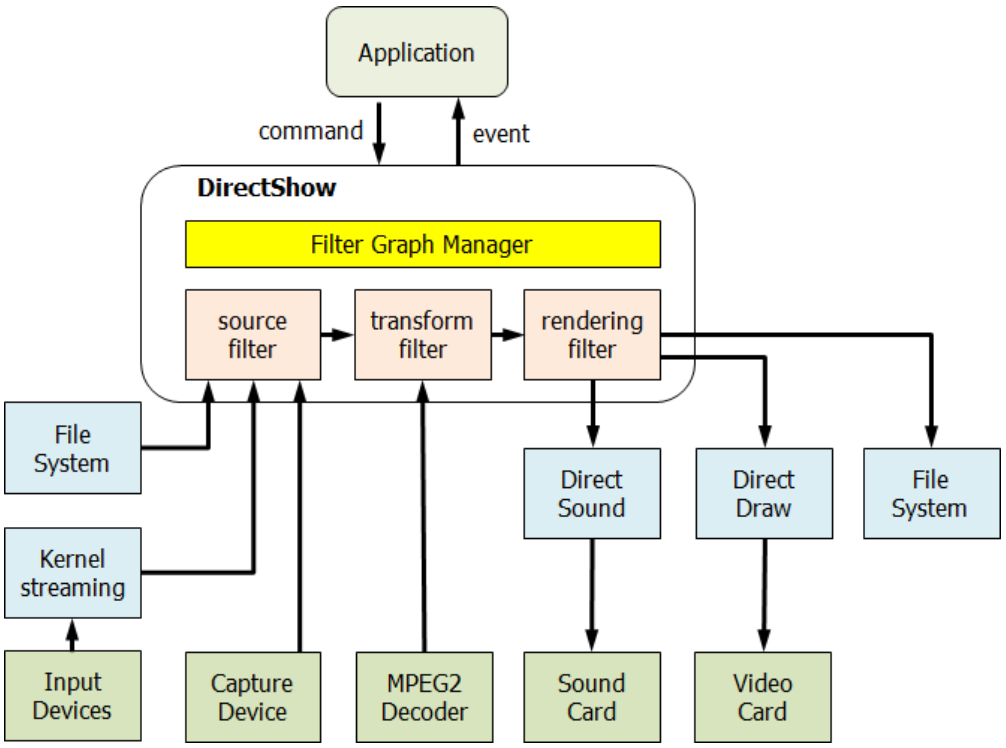


Figure 4. The internal architecture of the DirectShow library

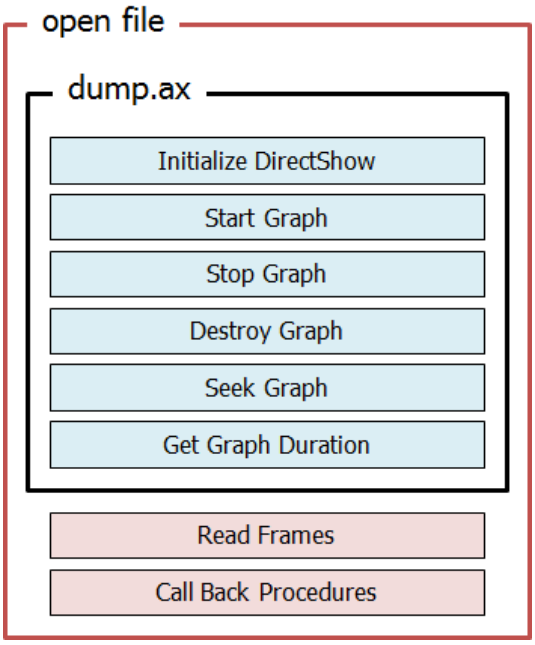
To perform encoding operations, it requires audio and video information from the input image. In our system, we use the DirectShow library [9] for these low-level input operations. The DirectShow library is not a simple module, but a set of filters based on the DirectShow specific data structures. In this framework, it works with source filters, transform filters, rendering filters, and various device drivers with the filter graph manager concept, as shown in Figure 4.

Filters in DirectShow can be classified into the following three types:

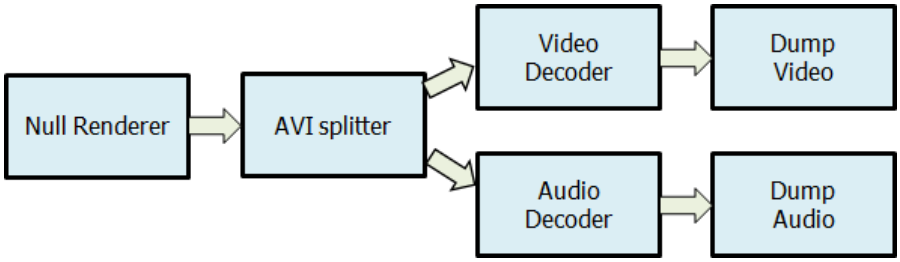
- **Source filters**: read the contents of media files or input devices such as digital cameras. Network transmissions are also available. Basically, these filters read a multimedia file on the disk or other storage devices, and extract the video and audio information.
- **Transform filters**: transform the input MPEG file from the source filter. These filters perform simple geometric transformations, image calibrations, sound effects, caption insertions, and others.
- **Renderer filters**: outputs audio signals and/or video sequences to the sound devices and/or video drivers. File output is also supported.

Figure 5 shows a typical implementation of multi-thread encoding operations with "dump.ax" filter, to extract the frame information from a sequence of video stream. To generate screen captions, we also use "ffdshow.ax" filter and "subtitles" filter to connect the video decoder and renderer filters. Our accelerated look-ahead processing based on the OpenCL-based GPU kernels achieves overall enhancements on the system processing speed. More detailed results are shown in the following section.

Copyright ⓒ 2014 SERSC



(a) overall architecture of "dump.ax" filter



(b) details of the "Initialize DirectShow" step

Figure 5. The internal architecture of the "dump.ax" filter

4. Results

Our system is implemented on a Windows PC with an Intel i7 CPU and GeForce GTX560 Ti GPU. We aimed at the most convenient user interface, and thus, we minimized the user input. Users can add the target video file with pressing the "+" button on the keyboard, or simply drag-and-dropping it to the program window, as shown in Figure 6. The target file will be shown on the list control of the main user interface window. At that time, it also displays the basic file information including file names, sizes, frame rates, and others. The right-most column shows the current status of the encoding process.

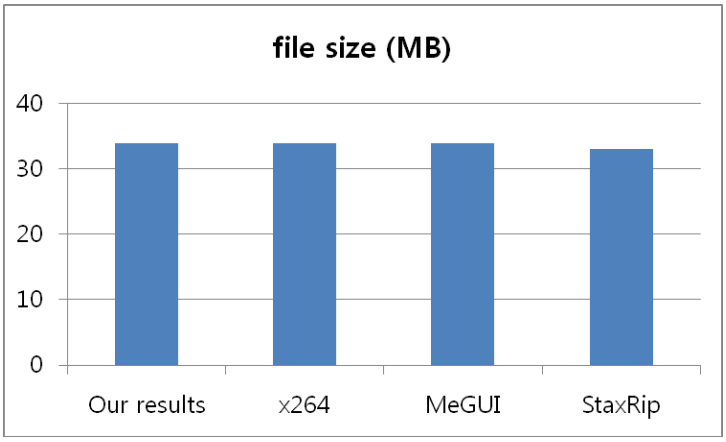
At the lower-left corner, a pair of arrows provides the facility of encoding order change. Below the arrows, a text-input window accepts user input for the output folder location. The default output folder location is the main background screen. Users can change this default location to any folder in the system. Users can also open the output folder to check the current folder status.

To check the execution speeds of a set of H.264 encoders, we use the same parameter settings as possible. Figure 2 shows the final benchmark results from a set of encoders, our

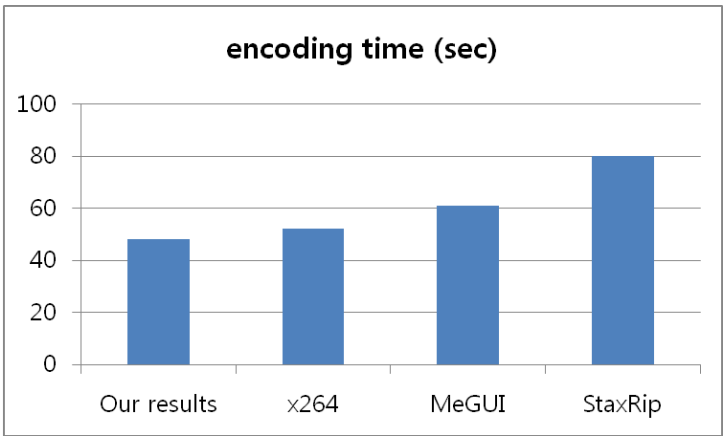
 Copyright ⓒ 2014 SERSC

implementation, x264 [8], MeGUI [10] and StaxRip [11]. Other encoders, x264, MeGUI, and StaxRip are selected from widely used H.264 encoders.

Figure 6. Our main user interface features: Red colored captions are added to the original captured image



(a) final file sizes



(b) elapsed encoding time

Figure 7. Experimental results from various implementations, for 154MB video file with 1920×1080 resolution.

As shown in Figure 7(a), for the example video file of 154MB with 1920x1080 resolution, the finally encoded file sizes are almost same sizes of about 34MB. In the case of encoding time, our implementation shows the best result. It shows about 50% enhancement in comparison with StaxRip. And, it shows at least 10% speed-up even in comparison with the previous x264 implementation.

5. Conclusion

We aimed to reduce the H.264 digital video encoding time, with GPGPU techniques. We started from analyzing the pros and cons of existing H.264 encoders, and located the heavy-computation areas in the real implementation. And, we applied parallel-processing techniques to those locations, while we preserve the already-optimized areas in the existing implementation. Finally, our OpenCL-based GPU kernel implementation achieves 10%-to-50% speed-ups in comparison with widely-used H.264 encoders.

Acknowledgments

This investigation was financially supported by Semiconductor Industry Collaborative Project between Kyungpook National University and Samsung Electronics Co. Ltd.

References

[1] T. Wiegand, *et al.*, "Overview of the H.264/AVC Video Coding Standard", IEEE Trans. on Circuits Systems for Video Technology, vol. 13, no. 7, **(2003),** pp. 560-576.
[2] S. Ghorbani and F. Zargari, "A Unified Architecture for Implementation of the Entire Transforms in the H.264/AVC Encoder", Int. J. of Multimedia and Ubiquitous Engineering, vol. 8, no. 1, **(2013)**, pp. 41–54.
[3] S. Lee and S. J. Park, "On improving the Fast Mode Decision of the Enhancement Layer in Scalable Video Coding extension of H.264/AVC", Int. J. of Control and Automation, vol. 5, no. 3, **(2012)**, pp. 207–216.
[4] A. Munshi, "The OpenCL Specification, version 1.0.29", Khronos OpenCL Working Group, **(2012).**
[5] NVIDIA, CUDA C Programming Guide, version 2.5, NVIDIA, **(2012),**
[6] N. Wu, M. Wen, H. Su, J. Ren and C. Zhang, "A Parallel H.264 Encoder with CUDA: Mapping and Evaluation", IEEE 18th Int'l Conf. on Parallel and Distributed System, **(2012).**
[7] E. Marth and G. Marcus, "Parallelization of the x264 encoder using OpenCL", ACM SIGGRAPH 2010 Posters Article No. 72, **(2012).**

 Copyright ⓒ 2014 SERSC

[8] L. Merritt and R. Vanam, "Improved Rate Control and Motion Estimation for H.264 Encoder", IEEE Int'l Conf. on Image Processing 2007, vol. 5, **(2007)**, pp. 309-312.
[9] A. Polinger, "Developing Microsoft Media Foundation Applications", Microsoft Press, **(2011).**
[10] MeGUI, http://sourceforge.net/projects/megui/.
[11] StaxRip, http://staxmedia.sourceforge.net/.

Authors

Young Chun Kwon

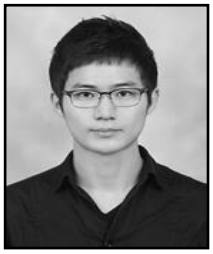
Young Chun Kwon was a master student in School of Computer Science and Engineering, Kyungpook National University. He is now a researcher in Samsung Electronics, Inc. His interests include computer graphics and parallel process.

Chanho Park

Chanho Park is now undergraduate student in School of Computer Science and Engineering, Kumoh National Institute of Technology. His interests include computer graphics and OpenCL.

Daeseok Oh

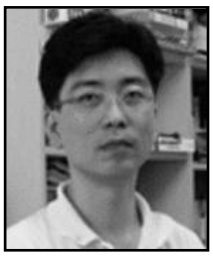
Daeseok Oh is now undergraduate student in School of Computer Science and Engineering, Kumoh National Institute of Technology. His interests include software engineering.



WooSuk Jang

Woosuk Jang is now undergraduate student in School of Computer Science and Engineering, Yeungnam University. His interests include network system.

Copyright ⓒ 2014 SERSC



Nakhoon Baek

Nakhoon Baek is currently an associate professor in the School of Computer Science and Engineering at Kyungpook National University, Korea. He received his B.A., M.S., and Ph.D. degrees in Computer Science from KAIST in 1990, 1992, and 1997, respectively. His research interests include real-time and mobile graphics.

Copyright ⓒ 2014 SERSC