System on Chip: H.264 Video Compression with PYNQ Z1

Michael Hicks, Christine Duong, Edward Hwang, Cory Kim, Jaemin Kim, Ryan Toeung, and Mohamed E. Aly
{mhicks, chd, ethwang, coryk, jaeminkim, rktoeung, mealy}@cpp.edu
Electrical and Computer Engineering Department
California State Polytechnic University, Pomona
Pomona, California

***Abstract*— People are constantly communicating and sharing information. To share large amounts of data, people have found ways to compress data to be sent or shared over distance. Videos have been used in many disciplines to share information. An example is in learning environments such as a school. People could post a video on a website, but there it might not support the file size. The video would need to be compressed to a smaller size to be available for sharing. Video compression is data compression. Data compression is used to minimize the size of data being stored/communicated. In this paper, video compression was studied. Compressing videos has a long history. There have been various algorithms: H.261, H.263, MPEG and more. The chosen algorithm was the H.264 video compression. Previous research aided the study and tests of the algorithm. Video compression is dependent on finding patterns within the data and compacting the original data. This eliminates redundancy and irrelevancy in the compression process and allows for smaller file size. The algorithm was tested on the PYNQ Z1's onboard ARM processor. Using Cisco's open source H.264 algorithm, the algorithm was able to be implemented on the PYNQ Z1's onboard Linux. With the success of building the OpenH264 codec library, performance metrics were tested on the encoder portion of the H.264 since the focus was video compression. The test files were in raw .YUV format. The testing involved gathering data on total frames to encode, target bitrate, time to encode, frames to encode per second, input and output resolution, .YUV file size and .264 file size.**

Keywords: Xilinx, PYNQ Z1, Linux, System on Chip, H.264, PuTTY, video, compression, frames, bitrate, bitstream, encoding, decoding, codec, OpenH264, Discrete Cosine Transform (DCT)

I. Introduction

When looking at the history of the H.264 video compression, we can see that it dates way back. H.264 video compression has been an industry standard for years and does not seem to have changed. This is due to the fact that when the video is running through a H.264 compression, it gets formatted into a format that takes up less capacity when it is stored or transmitted. Running through a H.264 compression, the video runs through both an encoder and a decoder. First off, the encoder converts the video in a compressed format, which would then call onto the decoder. The decoder would then convert the compressed video back into an uncompressed format to allow for the end user to view the video in normal form. The image below depicts how the video encoding and video decoding works.

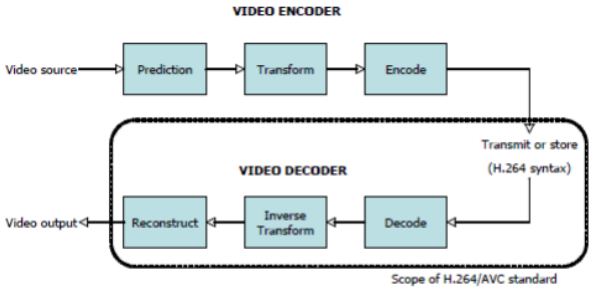

Fig. 1 H.264 Encoder and Decoder Overview [1]

A brief overview of how the H.264 codec works is when the video is running through the encoding phase, it would first carry out a prediction process. Then it would transform and ultimately end up encoding to produce a compressed H.264 bitstream. Once it runs into the decoder, the decoder would then use an inverse transformation process. Afterwards, the decoder would reconstruct the video to produce a decoded video sequence.

We can see this being used in practice every single day. One example that shows the power of H.264 video compression is seen in a single-layered DVD. If the DVD disk is formatted in an MPEG-2 format, then it can hold about 2 hours' worth of video on the disk. However, if it were to be converted into H.254, the disk would be able to hold 4 hours or more onto that disk. Other than DVD's, products such as High Definition Televisions, Apple products, such as iTunes videos, internet videos, such as YouTube videos, and video conferences all go through H.264 encoding and decoding.

II. Related Work

A. History

There are two standardization bodies that contributed in shaping and developing standardization for the media industry: International Organization for Standardization (ISO) and International Telecommunications Union (ITU). The ISO and International Electrotechnical Commission (IEC) created Motion Picture Experts Group (MPEG) that established one of the major video compression standards that are widely used. There are numerous variations of the MPEG variation. The first one is MPEG-1 part-2 and it was standardized in 1993. It was developed for "video and audio storage on CD-ROMs. It

supports .YUV 4:2:0 at common intermediate format at 352x288 resolution and the motion vectors were coded with a lossless algorithm [2].

The successor of MPEG-1 part-2 is cleverly named as MPEG-2 part-2. It was standardized in 1995 and supports video on DVDs, standard definition TVs and high definition TVs (HDTVs). In addition to .YUV 4:2:0, it also supports .YUV 4:2:2 formats. This standardization supports interlace and progressively scanned pictures. It introduced "profiles and levels to define the various capabilities of the standard as subsets" [2]. Lastly, it had scalable extensions which permit the division of a continuous video signals into two or more coded bitstreams representing the video at different resolutions, picture quality, or picture rates. Other features include data portioning, non-linear quantization, VLC tables and improved mismatch control.

Then came MPEG-4 part-2 which was standardized in 1999. It supports video on low bitrate multimedia applications on mobile platforms and the Internet. It also supported object-based or content-based coding where a video scene is coded as a set of foregrounds, background objects, coding of synthetic video and audio including animation. It shares a subset with H.263, which will be discussed later. The last variation of MPEG that will be discussed is MPEG-4 part 10, which was standardized in 2003 and was co-published as H.264 Advanced Video Coding (AVC), which will be discussed later.

The second widely known video compression algorithm was developed by the International Telecommunications Union (ITU). They developed the H.26x line of video compression standards. The first standardization named H.261 was standardized in 1988 and was developed for video conferencing over Integrated Services Digital Network (ISDN). It supported "Common Intermediate Format (CIF) and Quarter Common Intermediate Format (QCIF) resolutions in .YUV 4:2:0 format" [2]. It used block-based hybrid coding with integer pixel motion compensation. After H.261, H.262 was standardized as MPEG-2 part-2 in 1995.

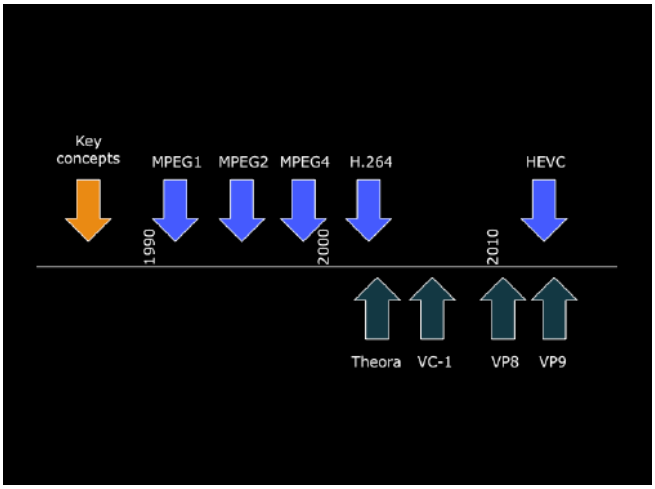

Fig. 2 Timeline for various video coding standards and formats [2]

The H.263 and H.263+ were standardized in 1996 and 1998, respectively. These standards improved quality compared to H.261 at lower bitrate to enable video conferencing and telephony. It uses "sub-pixel motion vectors to 1/8th pixel accuracy for improved compression., and shares subset with MPEG-4 part 2" [2].

The H.264 AVC was standardized in 2003 and supports video on the Internet, computers, mobile, and HDTVs. It significantly improved the picture quality compared to H.263 at low bitrates, but at the cost of increased computational complexity. It also improved "motion compensation with variable block-size, multiple reference frames and weighted prediction and implemented in-loop deblocking filter to reduce block discontinuities" [2].

The H.265 or High Efficiency Video Coding (HEVC) was standardized in 2013 and consists of a similar basic structure as H.264 AVC. The H.265 supports ultra HD video up to 8K resolutions with frame rates up to 120 frames per second (fps). It offered "greater flexibility in prediction modes and transform block sizes and used more sophisticated interpolation and deblocking filters" [2]. The H.265 is more efficient when compared to H.264 in terms of bitrate savings for the same picture quality.

III. Mathematical Work

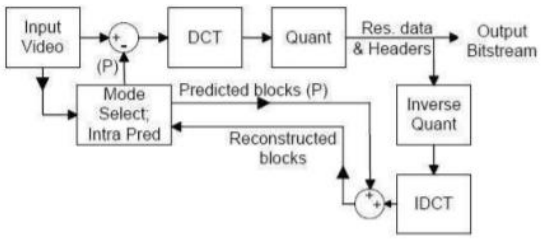

Fig. 3 Mathematical Flow Chart (Using Intra-Prediction) [3]

To produce an encoded video, the math behind the H.264 codec utilizes prediction, transform (DCT/DWT) and quantization to create a compressed file.

A. Prediction

There are two types of prediction, intra prediction and inter prediction. "Intra prediction utilizes spatial correlation in each frame to reduce the amount of transmission data necessary to represent the picture," [3]. It uses block sizes of 16x16 and 4x4 blocks only. Intra prediction uses previously coded data from the current frame's surrounding and pixels to come up with a prediction. On the other side, inter prediction bases the result from "other frames that have already been coded and transmitted," [1]. In inter prediction, the block sizes can range from 16 x 16 down to 4 x 4. A prediction is formed by taking a macroblock of a frame and using one of the prediction methods. The frames that have been coded may "occur before or after the current frame in display order," [1]. A macroblock is a unit describing a frame of 16x16 pixels. After getting the prediction, it is subtracted from the current macroblock to create a residual block. The residual block is a section of the frame that has been encoded.

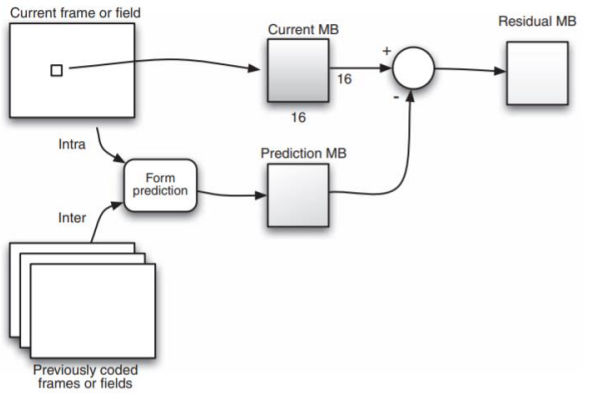

Fig. 3 Prediction Flow Diagram [1]

B. Discrete Cosine Transformation

The discrete cosine transformation (DCT) allows an image to be separated into parts during the encoding process. Each part contains weights of importance for a standard basis pattern, in respect to image quality, similar to how a Fourier Transformation converts a signal to the frequency domain in weights of harmonics [3]. H.264 codecs implement an approximate form of the DCT through a 4x4 integer transform of the residual blocks that were acquired, minimizing the data size and the computational complexity required to output a set of coefficients that indicate the weights of a standard basis pattern [4]. When combined through a DCT inverse transform, the 4x4 standard basis patterns and 4x4 coefficient weights recreate the encoded 4x4 image block.

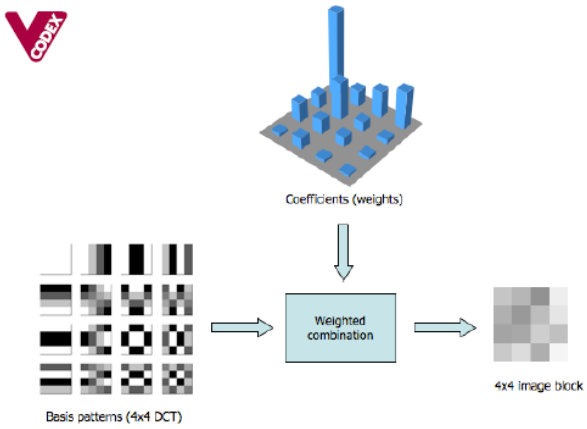

Fig. 4 Example of an inverse transform [4]

C. Quantization

From the block of transform coefficients, the output is quantized by dividing each coefficient by an integer. Quantization "reduces the precision of the transform coefficients according to a quantization parameter," [1]. The quantization parameter has 52 levels possible, which explains the effect of the compression and quality of a video.

After the block has been quantized, the resulting block will typically be left with "most or all of the coefficients are zero, with a few non-zero coefficients," [1]. A high quantization parameter value will set more of the coefficients to zero. The more coefficients equal to zero result in "in high[er] compression at the expense of poor[er] decoded image quality," [1]. On the other hand, a low quantization parameter will result "in better image quality at the decoder but also in lower compression," [1]. Thus, the quantization parameter describes the relationship between compression and quality.

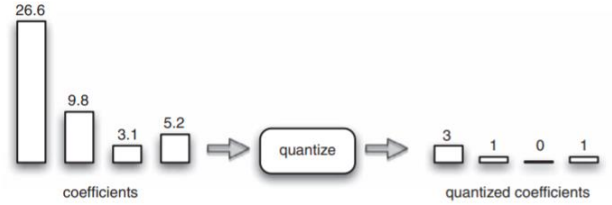

Fig. 5 Example of Quantization [1]

D. Bitstream Encoding

The video is converted into "a number of values that must be encoded to form the compressed bitstream," [1]. The values that will be encoded are "quantized transform coefficients, information to enable the decoder to re-create the prediction, information about the structure of the compressed data and the compression tools used during encoding, information about the complete video sequence," [1]. The encoder creates a bitstream which is machine language (or binary code) of the mentioned values for the computer to understand. The bitstream is made by using variable length coding and/or arithmetic coding. The encoding process compact binary information can be stored or transmitted for decoding or other simply the need for a smaller file size.

IV. LIMITATIONS

Due to the primary constraint of time, from the coronavirus becoming a worldwide pandemic to the limited research time available, the project was not able to fulfill all its goals within the 15-week timeframe. Initially, the project was to be run on a Linux virtual machine utilizing Jupyter that will execute scripts on the PYNQ-Z1, but due to operating system incompatibilities with Windows 10, the Linux virtual machine did not run properly. To resolve this, the Linux virtual machine was discontinued, and PuTTY was directly installed on Windows 10 to communicate with the board. In addition, while successfully implementing the H.264 source code onto the onboard ARM processor, creating an FPGA overlay architecture from scratch required additional time to research, implement, and test onto the PYNQ-Z1. The next problem that arose were the shipping slowdowns that arose amidst the coronavirus pandemic. A Raspberry Pi 3 was to be tested for performance alongside the PYNQ-Z1 once it was purchased online, shipped, and arrived by the tenth week of the project, no longer possible through online retailers in a timely manner. Lastly, the PYNQ-Z1's usable storage was constrained to about 500MB, limiting the video inputs available for compression to 1080p rather than larger resolutions such as 4K video.

V. Experimental Data and Results

Video Name	Input Resolution	Total Frames to Encode	Target Bitrate	Time to Encode (s)	Frames Encoded Per Second	Output Resolution	.YUV File Size	.264 File Size
akiyo_cif.yuv	352 x 288	300	6000	2.571902	116.645191	352 x 288	44.55 MB	0.657 MB
coastguard_cif.yuv	352 x 288	299	6000	5.389867	22.660001	356 x 288	44.55 MB	0.734 MB
flower_cif.yuv	352 x 288	246	6000	4.345483	56.610508	357 x 288	37.125 MB	0.621 MB
highway_cif.yuv	352 x 288	1807	6000	26.517655	68.14328	358 x 288	297 MB	5.254 MB
news_cif.yuv	352 x 288	300	6000	3.277382	91.536476	359 x 288	44.55 MB	0.718 MB
waterfall_cif.yuv	352 x 288	260	6000	4.348032	59.797168	360 x 288	38.61 MB	0.661 MB
dog.yuv	1920 x 1080	299	65000	207.269664	1.437741	1920 x 1080	911 MB	3.961 MB

Fig. 6 Data Table

After running the Open H.264 on our onboard Linux, we were able to modify the files and begin testing. In order to record data for performance evaluation, we found raw .yuv video files to compress and measure the changes into .264 encoded video [6]. The metrics that we focused on during the comparison were the time it took to encode the file, the frames encoded per second, and the output file size.

In our first comparison, akiyo_cif.yuv, the original file size had a resolution of 352x288 with a total of 300 frames that needed to be encoded. Within the configuration files, we determined the target bitrate to be 6000 and the output resolution to stay the same. After running the encoder, it took about 2.57 seconds to encode the 300 frames of the original file at a rate of 116.64 frames per second. After observing the outputted .264 file, we could also see the drastic decrease in file size. The original .yuv file was 44.55MB and after encoding, the .264 was only 0.657MB. The total compression was approximately 98.52%.

We took more tests with .yuv files of the same resolution size and target bitrate, but with a similar number of frames to encode. The data we recorded followed the same pattern as the results from the compression of akiyo_cif.yuv, a near 98% compression rate with slight variations in frame encoding speed. The only file that had a differing result was highway_cif.yuv, as it had 1807 frames to encode, and took 26.5 seconds to encode all its frames.

In the last data test, we tried a file, dog.yuv, of a much bigger resolution, 1920x1080 [7]. In our initial test with the dog.yuv, we left the target bit rate at 6000. However, after viewing the output .264 file we saw that it was clearly not enough bitrate as not all of the 299 frames were encoded. We slowly increased the target bit rate to 65000 where all 299 frames were encoded, and the output file was visually of similar quality to the original video. As a result of the large change in size, the time it took the board to encode shot up to nearly three and a half minutes, encoding at a rate of 1.43 frames per second. The .264 file size ended up being 3.96MB in comparison to the original .yuv file at 911MB, with a total of 99.56% compression.

VI. Conclusion

To conclude, this project of video compression dealing with H.264 was a very informative and interesting topic to research and develop. With an H.264 video compression, it would run through two different states. First off, the video would need to be sent through an encoder. What the encoder does is essentially take the inputted video and compress the video and produce a bitstream to allow for the next step to happen. This next step would be in the decoding section. The decoder is the latter half of the video compression. Once the encoder finishes compressing the video and passes through the bitstream, the decoder will then start to go to work. The decoder will inverse transform the compressed video and essentially reconstruct it back into a form where the video is easily accessible.

During the semester, we progressed tremendously throughout this project as we started with no prior knowledge to this topic whatsoever. First off, just to understand what this topic was about and how we would tackle the project. At this point in time, we started doing our research on the topic of H.264 video compression. With the help of our supervising professor, Mohamed El-Hadedy, we decided to implement this system on the PYNQ-Z1 FPGA Board. Next, after understanding how H.264 Video Compression works and having a basis on where our project was going to be performing, we continued with research. However, not on how the compression system works, but more on how to implement the video compression code onto the PYNQ Board.

After understanding how the whole system works, we then started the implementation portion. After a few trial and errors, we ended up using the program PuTTY. PuTTY allows for us to hook up the PYNQ-ZQ Board to run Linux. Once Linux is was running, we are able to implement our C Code that would perform the video compression. Afterwards, we are then able to insert videos to perform compression. This program prefers the file type of .yuv, so that is exactly what we provided it. In the data and results portion, the data collected is shown.

Due to the nature of the project and the time consumed trying to first understand and implement the whole thing, as a group, we were only able to implement the encoding portion. However, with this implementation, we got a great understanding of this whole thing. We learned many new skills, such as how to implement Linux through PuTTY, run C Code on a Linux platform, push specific files through PuTTY and onto the board to perform the C Code, and retrieve the compressed video file from the board. We are also able to see the data that has been collected through text file that were provided. Accumulating this all together, this project has been a huge success and we all now have a greater understanding of these topics.

To test the PYNQ Z1's performance, a Raspberry Pi 3 can be used to get better metrics. Both devices carry an ARM processor, Dual ARM® Cortex™-A9 MPCore™ with CoreSight and 1.2 GHZ quad-core ARM Cortex A53 for the PYNQ Z1 and Raspberry Pi 3, respectively. The PYNQ Z1 offers 512 MB of DDR3 for RAM. The Raspberry Pi 3 has 1 GB of DDR2 for RAM. These two devices are close in approximation, and both offer Linux as an operating system. When it comes to gathering more test data on the PYNQ Z1, the Raspberry Pi 3 will provide better performance metrics.

Another tool to explore is the Xilinx tool to create an overlay architecture. The overlay architecture can be used to increase the speed up of the process of compression. Using certain benchmarks and Amdahl's Law, the portion of code that is the slowest can be adapted to "virtual" hardware. With overlay architecture, it can be tested to see the improved performance of the encoding process.

ACKNOWLEDGMENTS

We thank Dr. Mohamed E. Aly for his support throughout this project. This work was performed with Cisco's open source H.264 codec library: OpenH264.

REFERENCES

[1] I. E. G. Richardson, The H.264 advanced video compression standard, 2nd ed. Hoboken, NJ: Wiley, 2010.

[2] I. E. G. Richardson, "Historical Timeline of Video Coding Standard and Formats," VCodex. [Online], Available: https://www.vcodex.com/historical-timeline-of-video-coding-standards-and-formats/

[3] Lee, M. and Moore, A., 2006. H.264 Encoder Design. [Online] Available: http://csg.csail.mit.edu/6.375/6_375_2006_www/projects/group3-report.pdf [Accessed April 25, 2020].

[4] D. Marshall, "The Discrete Cosine Transform (DCT)." Cardiff School of Computer Science & Informatics, Oct. 2001. [Online], Available: https://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/node231.html [Accessed April 25, 2020].

[5] I. E. G. Richardson, "H.264/AVC 4x4 Transform and Quantization," VCodex. [Online], Available: https://www.vcodex.com/h264avc-4x4-transform-and-quantization/ [Accessed April 25, 2020].

[6] "Xiph.org Video Test Media [derf's collection]," Xiph. [Online], Available: https://media.xiph.org/video/derf/ [Accessed April 10, 2020].

[7] "UVG Dataset," Ultra Video Group. [Online]. Available: http://ultravideo.cs.tut.fi/#testsequences. [Accessed: 26-Apr-2020].

Michael Hicks was born in Lake Havasu, Arizona on May 19th, 1988 and currently resides in the city of Yucaipa, California. After receiving his associate degree in Mathematics at Crafton Hills Community College in Yucaipa California in 2017, he transferred to California State Polytechnic University, Pomona, California, which he will receive a B.S. in computer engineering in December 2020.

Throughout his time at Cal Poly Pomona he studied FPGAs and Verilog HDL, and Verilog VHDL. Current projects that are ongoing are linked into FPGA data and video compression, Arduino programming, and C# programming for FAT16/32 systems.

Ryan K. Toeung was born in Montebello, California in 1997. He graduated from Rosemead High School in 2015. He began his college education at Pasadena City College and graduated with an AA degree in Math and Science. He transferred to California State Polytechnic University, Pomona, where he is currently working towards an undergraduate degree for a Bachelor of Science for Computer Engineering. Upon graduating with an undergraduate degree, he plans to work with software and hardware development.

Christine Duong was born in Fountain Valley, California in 1999. While attending Arnold O. Beckman High School, she took joint classes at Irvine Valley College. In May 2017, she acquired her high school diploma and AA degree. In Fall 2017, she began her first year at California State Polytechnic University, Pomona. She is currently working on her undergraduate degree, for a Bachelor of Science in Computer Engineering. For future plans after graduating, she aims to work in software-related roles and pursue higher education.

Cory Kim was born in Irvine, California in 1999. After graduating from Irvine High School in 2017, he began studying at California State Polytechnic University, Pomona, to obtain an undergraduate degree for a Bachelor of Science in Computer Engineering. His current plans are to complete his degree while working with ongoing projects in programming microcontrollers and FPGA HDL development.

Edward Hwang was born in San Gabriel, California in 1999. He graduated from Walnut High School in May 2017 and was accepted to study at California State Polytechnic University, Pomona for an undergraduate degree for a Bachelor of Science in Computer Engineering. He is hoping to graduate in May 2021 and continue to work with projects involving FPGA.

Jaemin Kim was born in Seoul, South Korea in 1994. After graduating from Deer Creek High School located at Oklahoma in 2012, he moved to California to start his higher education. He completed his Associates in Cerritos College, receiving Associate's Degrees in Computer Science and Physics. After completing his Associate's Degrees, he transferred to California State Polytechnic University, Pomona where he is earning a Bachelor of Science in Computer Engineering. Upon graduating, he plans on working in integrated circuit design.