Parallel H.264 Video Encoding and Decoding

Steven Wirsz

10 December 2015

Sequential Implementation

Video codecs, or encoding/decoding algorithms, are absolutely essential for the realistic use of video in any digital device, whether it be computers, tablets, cell phones, streaming video devices, digital cameras, or countless hybrid devices. Uncompressed video is simply too insanely large to handle. If just common 1080p video is considered, the uncompressed data rate would be calculated by: color depth * vertical resolution * horizontal resolution * refresh frequency = 24*1920*1080*60=2.98 Gbit/s. Modern codecs such as H.264 can easily reduce this to 2 Mbit/sec, a compression rate of more than 1000 times without significant loss in quality as far as a human eye can perceive. This permits transmission pipelines and video file sizes to be similarly reduced by a factor of 1000.

The most popular video encoding method, H.264 goes under several names. It is also referred to as x264, MPEG-4 Part 10, AVC or Advanced Video Coding. It is a revolutionary standard that was completed in 2003 and lasted for an entire decade as the definitive video encoding technology before recently being superseded by H.265. Encoding efficiency, as determined by the trade-off of bit rate to video quality, improved by a factor of 4X to 10X above the most effective predecessor codecs of the time. Just like Google's idea of incoming links showing the value of a webpage revolutionized Internet searches, the revolutionary idea behind H.264 is the concept that fragmented parts of images, known as macro blocks, could be matched up to different locations within nearby video frames to greatly reduce redundant information and the resulting video size.

Implementation of H.264 breaks down a sequence of image frames into three distinct types, with typically 30 to 60 frames encoded per second of video. The first type is I-frames or image frames, simply stand-alone single images encoded by the JPEG standard. These act as reference frames for the other frames, and are also used as breakpoints. Only at I-frames can video sequences be split or edited, since they are the only frames that stand alone by themselves. P-frames are forward predicted frames,

frames that can be built by referencing only previous I-frame and P-frames. B-frames are bidirectionally predicted frames that may be constructed from all nearby frame types, both frames sequentially before and after that image. The greater the ratio of B-frames to I-frames, the greater the video compression, but greater the CPU load and more difficult "jumping to different locations" in playback becomes. The sequence of both encoding and decoding sequential images in H.264 does not follow chronological order, but follows an order based on these dependencies. When a group of images is selected, first I-frames are encoded at the beginning and end of the sequence, the P-frames are then generated using motion estimation derived from the first I-frame and earlier P-frames, and then finally the B-frames are created using any of the surrounding frames. A typical sequence may be something such as IBPBPBPBPI. (Shoham)

All images are ultimately encoded with the Jpeg standard, however usually I-frames are the only frames with significant detail, requiring significantly more effort in terms of CPU usage, memory size, and data transfer during the image compression step. The Jpeg algorithm compresses a single image using a combination of different algorithms to achieve a compression ratio of about 10 to 1 on average, and possibly as high as 30 to 1. The standard utilizes both lossless compression and lossy compression that throws out detail that the human eye has difficulty perceiving. Discarding imperceptible information and running each algorithm in sequence achieves a much higher compression rate than any single technique alone would achieve. The first step in JPEG compression is converting typically 4-bit RGB color data (3 bits for red, green, blue intensity, 1 bit for luminosity) into 3-bit YUV/YCC. (2 bits for color/chroma data) Using a color table, color values are scaled and rounded while luminosity values which the human eye can perceive to a greater degree are maintained unaltered. (Wikipedia)

The next step is performing a discrete cosine transform. Because human images typically have pixels of similar properties adjacent to each other in an image, a blue sky for example, the image is

divided up into blocks sizing between 4x4 to 16x16 pixels and only the top left corner pixel is stored as its normal value. All other pixels in the blocker transformed to a derivative, + or - their relative value to the top left pixel of the block. To try to maximize sequences of similar pixel values which might have similar numbers, each block is read into an array in a diagnosis zig-zag pattern instead of simply by rows or columns, which tends to more often find repetition. (Wikipedia)

Discrete cosine transforms and inverse discrete cosine transforms are computationally simple to do as they can be performed by a processor using only addition, a much faster operation than most others, but the vast number of computations required can consume as much as 30% of the processor cycles when decoding video. A typical low-resolution video 352x288 at 30 frames per second may perform as many as 71,000 inverse DCTs per second, the actual number being highly dependent upon video content) Independent of other parallelization, the negligible memory requirements of DCT/IDCT and enormous number of competitions make them ideal for task parallelization using specialized dedicated hardware coprocessors. (Shoham)

The next step is quantizing DCT values. This is also done in to take advantage of the relative perceptual ability of the human eye. High frequency values in the DCT table, corresponding to small changes in the image, are rounded because the human eye does not perceive small, low contrast differences. Low frequencies, corresponding to large features or high contrast edges in the image are maintained unchanged. (Wikipedia) Dequantization may require anywhere from 3% to about 15% of the processor cycles spent in a video decoding application. Like the DCT and IDCT, the memory requirements of quantization and dequantization are typically negligible and can also be outsourced to coprocessors. (Shoham)

Finally two steps of lossless compression are applied. First the algorithm applies run length encoding, a method of summarizing sequences of perfectly duplicate values of pixels, then a second step

of either Huffman coding or arithmetic encoding is used to compress image further. Huffman encoding is less CPU intensive, building probability tables for repeated sequences, and then using prefix free codes to allow more frequent occurrence of characters to take up less bits than less frequent sequences. The other possibility is using arithmetic encoding (AC) that stores ASCII data within floating-point numbers. AC achieves higher compression rates because of avoiding the loss of space utilization of prefix coding, but multiplying and dividing are much more CPU intensive tasks and slows the entire video decoding process significantly. Since high definition and 4K video require so much CPU usage in the following steps, AC encoding as the variable length encoding step is becoming more standardized. (Wikipedia)

Performance is typically demanded more often from video decoding for smooth playback rather than for video encoding. Unfortunately, lossless/variable length decoding is much more computationally demanding than variable-length coding. Encoding performs one table lookup per sequence (where a sequence is encoded using multiple bits). Decoding requires a table lookup and some simple decision making to be applied for each bit, on average, about 11 operations per input bit. For these reasons, processing requirements for decoding video are proportional to the videos bit rate and can consume as much as 25% of the processor cycles, but is a negligible factor when encoding. (Shoham)

Both the most tweaked and highest source of compression achieved by H.264 is from motion estimation/motion compensation when creating P-frames and B-frames. This process boosts the compression ratio from JPEG's 10 or 30-to-1 up to a range of 200 to 1000-to-1. The image space is again divided up into 16x16 regions, each square or 'macro block' processed independently. The algorithm for motion estimation takes these macro blocks one at a time and attempts to compare them to 16x16 regions on other frames, known as reference frames, to find close matches. The algorithm stores the identity of the reference frame and the X/Y offset or difference in position (called a motion vector)

between the two macro blocks, and calculates a pixel by pixel difference using either the sum of absolute differences (SAD), or sum of squared differences (SSD). Only the most closely matched reference is maintained, and then stored as a difference frame (or a prediction error frame) Later when that decompression routine runs motion compensation, combining the reference frame with the stored differences will reconstruct the original image. Since the stored differences contain very little unique information, Jpeg compression is able to reduce the size of the resulting P/B-frame to only a fraction of the size of what the original I-frame would be. (Shoham)

Parallel Implementation

When encoding video, a complete and exhaustive search of all surrounding possible motion vectors within surrounding frames has no dependency requirements and could be performed in parallel very easily. Unfortunately full search is computationally impossible, requiring a prohibitive 4.6 billion arithmetic operations per second for just a low-resolution video at 352x288 and 15 frames per second, and 810 billion arithmetic operations per second for 4K video. Actual implemented techniques of limiting motion estimation search are extremely complicated and retail encoders optimize their methodologies over time, resulting in significantly different speed and compression rates among H.264 encoding programs in practice. (Shoham)

Several simple motion estimation techniques exist, but their use creates dependencies and limits the ability to run motion estimation searches in parallel. One well known technique is 5-log. Five different candidate motion vectors are taken from different areas of the image to evaluate the best possible SSD. Unlikely areas are discarded, and five new motion vectors are picked close to the last best match. This is repeated until the search area narrows to a fractional pixel distance and then the best possible motion vector match found in the entire process is saved. Since this search technique by itself has high dependency by requiring the previous results to be tested before the next search is performed, variations on this, such as narrowing down to the N best search areas instead of just the best search area, or performing more than 5 parallel searches are used to take advantage of the exact number of parallel cores available. (Shoham)

Beside task-level (as mentioned before with coprocessors) and the motion estimation searching described above, numerous other ways of both encoding and decoding H.264 on parallel processes exist. CPU usage for encoding H.264 video is completely dominated by motion estimation, requiring for an average video around 95% of arithmetic CPU usage, 90% of the control instructions, and 96% of the

data transfer between cores. The decoding process is much more variable and CPU usage is divided up between several different processes.

Again, task level parallelism is only useful for video decoding and breaks down the decoding process into each functional task: inverse quantization, inverse transform, motion compensation, synchronizing and parsing, entropy decoding, and motion compensation. (Krishnan) The main problem with task level decomposition is load-balancing and scalability. The number of instructions and time required to execute any particular class is determined primarily by the data being processed, and cannot be predicted in advance. The limited number of different tasks also makes scaling the process to more than a few cores very difficult as well. (Meenderinck) For this reason although some type of task level parallelization occurs in many retail implementations, it is usually never used alone.

Four different data level decompositions exist, the Group-of-Pictures (GOP) level, frame-level, slice-level, and the macroblock-level. At the GOP level, a sandwich of frames, beginning and ending with an I-frame is assigned to a single processor. Ideally this would be coarse-grained parallelism, except for the requirement of B-frames to be bidirectionally constructed from all surrounding frames, including the 2 border I-frames. This means that each I-frame must either store the entire image data in a large volume of shared memory or be redundantly decoded by 2 processors. GOP parallelism also has poor scalability for certain activities such as streaming video. For applications like streaming, there are a limited number of groups of frames that can be decoded in parallel without either overtaxing the bandwidth of the channel or caching large sections of the data in advance before being delivered to the screen. (Krishnan)

Frame level parallelism was implemented for earlier versions of MPEG-4 compression, but has severe drawbacks when used with H.264. I-frames and P-frames must first be decoded, creating a sequence which cannot be parallelized and limiting the speed up factor according to Amdahl's law.

After this is done, conceivably the B-frames could be assigned to different processors to decode in parallel, except that the number of B-frames within a group of pictures (GOP) may be as few as two or three, which fails to utilize even the number of cores on a desktop CPU, much less the hundreds of parallel units of a typical GPU. B-frames can also be use each other as reference frames, meaning that compression rate suffers if B-frames are encoded simultaneously on independent processors. (Meenderinck)

Slice level parallelism is the practice of dividing up each image into independent segments that are processed in their own individual sets of I-frames, P-frames, and B-frames. This eliminates issues of dependency and ordering constraints that limit task, group, and frame level parallelism. A single image can be divided into n slices and encoded as if it was n different videos. Unfortunately scalability problems still occur as the encoder MUST know exactly how many parallel processors that the decoder will have in advance. For example, video encoded into 16 different slices will never be able to use more than 16 different processors with slice level parallelism for decoding. Unfortunately simply using a large number of slices is not a solution either. Slice level parallelism restricts the areas of each picture that are available for macro block prediction. This increases the bit rate required to maintain the same quality level. For one sample video, increasing from one slice per frame to 64 slices per frame increased the bit rate by 34%. (Meenderinck)

Macroblock level parallelism is the most effective and commonly used method available. Unfortunately, macro blocks introduce countless dependencies which must be taken into account as well. Motion vector prediction, intra prediction, and the deblocking filter use data from neighboring MBs and create very complicated dependencies. Truly independent macro blocks if they were overlaid on an image frame as it is encoded would visually appear to be points on a diagonal wavefront, as it moves from one corner of the image to the opposite diagonal corner.

Because of this architecture, the number of independent macro blocks capable of being independent processed varies considerably over time, starting with only 1 at the beginning of frame processing and also ending with 1. The resolution of the image also scales the number of independent macro blocks available. A low resolution picture may have only 6 independent macro blocks for 4 timeslots, but a high level frame (1920x1088) may have over 60 independent data blocks in over 9 timeslots. This is very convenient as processors and cores utilized for HD video successively are able to use more and more powerful processors with greater numbers of cores. (Meenderinck)

Data communication between different processors for macro block encoding can be limited to simply the results of computing the effectiveness of each motion vector. By only comparing the results of SSD calculations for example, the best possible motion vector can be determined and the other processors can simply discard their results. Unfortunately the process of allocating different motion estimation regions to different processors and calculating results independently loses some of the advantage and compression size gains caused by predicted motion vectors. This loss of compression measured as BDBR (Bjontegaard Delta BitRate) or the amount of bit-rate loss “pre-lossless compression” can be anything from an insignificant value for some images to values as high as 44% in other images. (Moriyoshi)

One other minor issue only affecting macro block parallelization is that the earlier stages of decoding the frame must be completed before parallel handling of macro block compensation. These stages, such as AC/Huffman decoding, dequantization, and inverse DCT must be completely computed first, but fortunately these activities can somewhat be internally be processed in parallel as well. In practice, these activities are performed similar to CPU pipelining, with a few processors being devoted to the early stages of N+2’s frame, motion compensation being performed on the majority of processors

for the N+1 frame, and finally the blocking and intra-prediction be performed for the N frame on yet 1 more core or processor. (Meenderinck)

One experiment shows that when utilizing SPMD on a GeForce GTX 580, which uses a CUDA (Compute Unified Device) Architecture of 16 streaming multiprocessors (SM), each having 32 processing elements/CUDA cores for a total of 512 total processing cores, a GPU accelerated encoder can run more than 10 times faster than a CPU implementation, with only a moderate increase in the bit rate or file size of the resulting file. (Moriyoshi)

Actual benchmarks and speed up rates for H.264 is again extremely dependent upon the actual data being encoded. For one test set of images, low-resolution video (352x288) experienced a decent speed up maximizing around 4.0 for 8 cores, but then drops to the factor of 3.5 due to communication overhead if 16 cores are utilized. High-definition video (1280x720) continues to increase moderately to the factor of 7.6 for 16 cores. Adding simply a second processor provides a speedup of 1.9 or an "efficiency" of roughly 90%. As additional cores are added, the efficiency drops considerably until for high-definition video, doubling the cores from 8 to 16 only adds a speedup of about 50%. (Baaklini)

Despite the decreasing gains in speed and loss of effective compression as more parallel processors are added, both H.264 encoding or decoding experience moderate levels of speed up from parallel processing. Moderate overhead, a minimum loss of compression rate, and slightly irregular CPU usage are only minor issues. The new H.265 standard with more explicitly defined support for parallel processors only makes incremental and evolutionary improvements on the features already provided by H.264, and can be thought of as only an extension to the existing standard, not a revolutionary new one. Since higher resolution video processing scales with additional processors at better efficiency as the image resolution increases, these parallel algorithms or variations on them undoubtedly will be used far into the future.

References

Baaklini, E., & Sbeity, H., & Niar, S. (2013). H.264 Parallel Optimization on Graphics Processors. *The Fifth International Conferences on Advances in Multimedia (MMEDIA), 2013* <http://hgpu.org/?p=9292>

Eyre, Jennifer. "Inside DSP on Digital Video: Processors for video—Know your options" 14 Mar 2005 BDTi 24 October 2015 <http://www.bdti.com/InsideDSP/2005/03/14/Idsp2>

JPEG. (n.d.). In Wikipedia online. Retrieved from https://en.wikipedia.org/wiki/JPEG

Krishnan, M., & Gangadharan, E., & Kumar. N. (2012). H.264 Motion Estimation and Applications, Video Compression, Dr. Amal Punchihewa (Ed.), ISBN: 978-953-51-0422-3, InTech, <http://www.intechopen.com/books/video-compression/h-264-motion-estimation-and-applications>

Meenderinck, C., & Azevedo, A., & Alvarez, M., & Juurlink, B., & Ramirez, A. Parallel Scalability of H.264. *CiteSeerX*. doi:10.1.1.119.957 <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.119.957>

Moriyoshi, T., & Fumiyo, T., Yuichi, N. (2011). GPU Acceleration of H.264 / MPEG-4 AVC Software Video Encoder. *APSIPA ASC 2011 Xi'an, 24 October 2015* <http://www.apsipa.org/proceedings_2011/pdf/APSIPA303.pdf>

Shoham, Amit. "Inside DSP on Video: Squeeze Play--How Video Compression Works" 29 Mar 2004 EE Times 24 October 2015 <http://www.eetimes.com/document.asp?doc_id=1272615>