18-551, Spring 2003
Digital Communication and Signal Processing Design
Group 15

Can You See Me Now?

(Implementation of H264 for use on cellular phones)

Wayne Chan (wkchan@andrew.cmu.edu),

Richard Duong (rqd@andrew.cmu.edu),

Charles Hamilton (charlesh@andrew.cmu.edu)

Table of Contents

Introduction 3

Background 3

The H.264 Video Compression Algorithm 6
- Motion Estimation 6
- Intra prediction 7
- Inter prediction 8
- Motion compensation 9
- Hadamard Transform and Quantization 9
- Context-Based Adaptive Binary Arithmetic Coding 10

Overall “Picture” 10

Approach 11

PCI and EVM intercommunications 14

Logistics 15

Results 16

Speed Issues 18

Obstacles to Overcome 19

Further Work 19

References 20

Introduction

With the increasing need to be connected in today's society, the demand for high bandwidth, feature rich cell phones has emerged. Cellular phones today now come with cameras and color screens that allow you to take pictures. However, cellular bandwidth is extremely costly and limited. In order to conserve bandwidth clever compression techniques are needed. Video is obviously the next step in cellular communication technology and therefore there needs to be a standard through which video is compressed. As of now there is no standard for cellular video communication.

Our group proposes to use the H.264 video compression standard to facilitate the transfer of streaming video over current cellular phone bandwidths. The emerging JVT/H.26L video coding standard proposed by the ITU-T video coding experts group (VCEG) and the ISO/IEC moving picture experts group is acceptable for a wide variety of bit rates, resolutions, qualities, and services. Because of these strengths, we feel that it is well suited to implement on the C67 EVM.

Background

The video coding standard H.264 achieves a high compression capability for a desired image quality. It is capable of the same quality video as previous video compression standards, MPEG-4 and H.263, with close to half the bandwidth usage. No single portion of the entire compression algorithm is drastically better than previous standards; however it is the combination of many small improvements that allow for such large improvements. Although the algorithm is not lossless, some techniques can be employed to attain highly efficient compression. Inter coding uses block-based inter prediction (through

time) and intra coding (within a single frame) uses spatial prediction. Motion vectors and intra prediction modes may be specified for a number of block sizes in the picture. This is then further compressed to remove spatial correlation using a transform, before it is quantized. This will keep a close approximation to source samples, while removing less important information. The two parts, the motion vectors or intra prediction modes and the quantized transform coefficient information are combined and encoded using either variable length codes or arithmetic coding.

Within the past 4 months, numerous companies have begun to implement the H.264 reference software on digital signal processors and field programmable gate arrays. Ingenient Technologies has a H.264 video compression algorithm running on its C64x DSP family. Equator Technologies similarly claimed its media processor's architecture would be capable of real-time H.264 encoding and decoding. Kevin Oerton of VideoLocus said his company's FPGA hardware acceleration encoder is capable of standard-definition video at 30 frames per second. Most recently, UB Video Inc. has produced a C64 implementation of the H.264 decoder to be used in real-time video applications. It is advertised as the world's first H.26L based video processing solution on Texas Instruments TMS320C64X digital media platform.

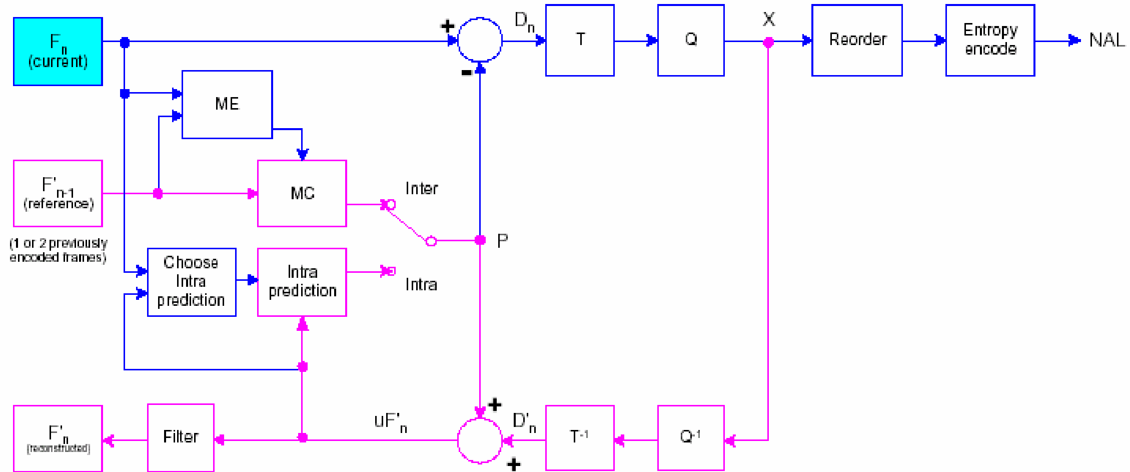
There has been a past 18-551 project having to do with video processing, i.e. the "Virtual Advertising" group. We will consider the methods used by this group to get video from the PC to the EVM and back again. Our group will look into using the MSDN libraries, vfw32.lib and winmm.lib, in converting the video into frames before sending it to the EVM, similar to the "Virtual Advertising" group. However, our project is different from this group because we are attempting to achieve a higher frame rate with a lower resolution. We are also implementing a more computationally demanding process than what has been previously implemented.

Preliminary work

We had a good deal of background learning and research to do in order to understand the entire algorithm, as well as the reference code before beginning the implementation. Our first step was to become familiar with the sophisticated algorithm through careful reading of the specifications and reference materials from online sources as well as from books. Next, we examined the available reference C code for encoding and decoding (available at: http://bs.hhi.de/~suehring/tml/). We tried to find the most computationally intensive parts of the algorithm, to determine what to tackle first, the encoder or the decoder. This took quite a bit of time because of the density of the material and the poor coding and commenting techniques used by the many writers. In the best interest of time, we decided to concentrate on the decoding portion of the software.

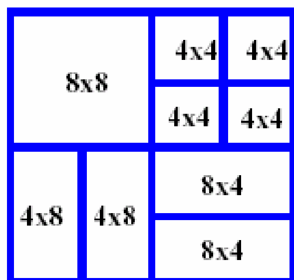
The Algorithm

Below is a block diagram of the H.264 encoding algorithm. The decoding process, shown in pink, is incorporated in the encoding process as prior encoded and reconstructed frames are used in the encoding process. Different modules within the encoding process include: motion estimation, motion compensation, Hadamard Transform, Quantizer, Entropy encoding, and a deblocking (loop) filter. Differences from MPEG-4, previous video compression technique, is the use of a Hadamard Transform as opposed to a Discrete Cosine Transform. The use of smaller sub pixel motion compensation. Also, the use of Context-Based Adaptive Binary Arithmetic Coding is a vast improvement to output bit stream size. A description of the process follows.



Block diagram of encoding process

First, an input frame is processed in units of a macroblock, which is a 16X16 pixel block of the original image. From here, a macroblock is subdivided further according to the prediction mode and motion content present in the macroblock.



The figure to the left shows the different possible subdivisions of a macroblock. It is referred to as the tree structure segmentation method. Motion vectors are transmitted for each block subdivision, so it follows that the smaller the subdivisions, the better the predicted outcome. However, with more to be transmitted, the overall computation time is slower.

Motion Estimation

H.264 does integer pixel motion estimation to evaluate the Sum of Absolute Difference (SAD) within a given search range. The SAD finds the sum of the pixel differences between the incoming block and the displaced block[1].

The macroblock is then either encoded using inter or intra prediction techniques based on the following equations:

If

$A < (SAD_{inter} - 2 * N_B)$

where

$$A = \sum_{i=1,j=1}^{16,16} |original - MB_{mean}| * (!\,(Alpha_{original} = =0))$$

and

$$SAD_{inter} = min(SAD_{16}\,(x,y),\, SAD_{k \times 8})$$

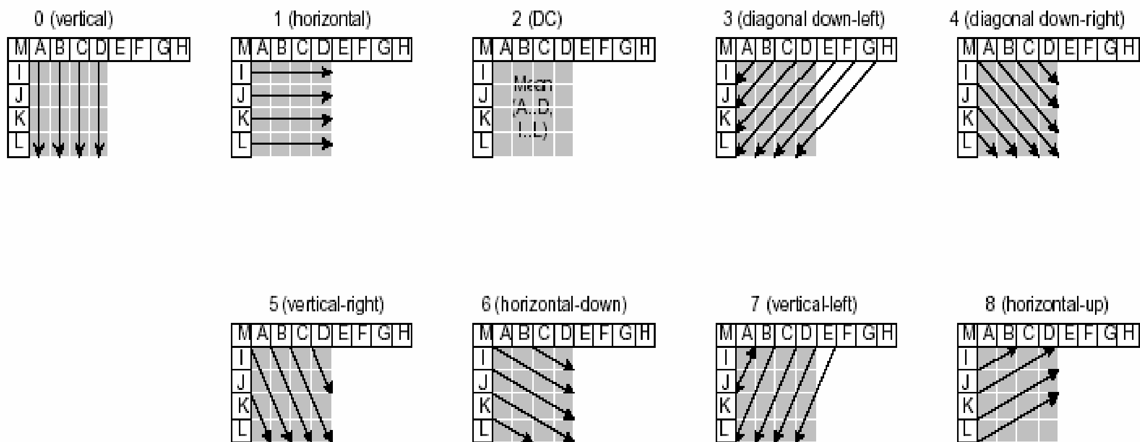
$$MB_{mean} = \left(\sum_{i=1,j=1}^{N_B} original \right) / N_B$$

N_B is the number of pixels in the block.

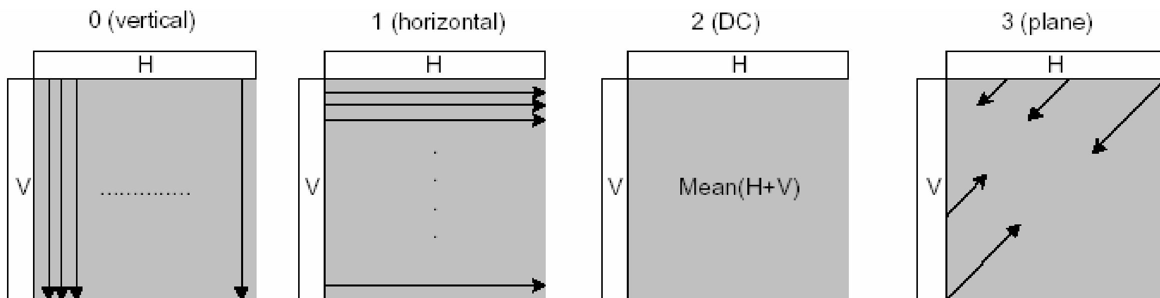
Then, INTRA mode is chosen, otherwise, INTER mode is used. [1]

Intra prediction

When intra prediction is chosen as a result of the above equations, the spatial redundancies of the video sample are exploited to create a prediction block. The resulting block, called an I-frame, only uses neighboring pixels to create a prediction frame. The pixels from the column to the left of and the row above the current block are the only ones considered in finding the best prediction. Below are the 9 modes of prediction for 4x4 luma blocks and the 4 modes of prediction for 16x16 blocks.



9 modes of prediction for 4x4 luma blocks



H.26L Intra 16x16 prediction modes (all predicted from pixels H and V)

Inter Prediction

Inter prediction uses previous and in some cases, future frames to create a prediction frame. The motion estimation and compensation techniques used take advantage of the temporal redundancies of consecutive frames. As shown in the macroblock subdivision graphic, for P-frames, the motion compensation can be done on block sizes of 16x16, 16x8, 8x16, and 8x8. Furthermore, if the block is split up into 8x8 blocks, these can be divided into 8x4, 4x8, or 4x4 samples. P-frames result from using one or more previously encoded and

reconstructed frame as reference frame(s) to create a prediction frame. B-frames (B stands for bi-directional) result from using previous P- or I-frames and future P-frames to create a prediction frame. Other types of frames, SP- and SI-, are used for switching between bit streams with similar content[3].

Motion Compensation

The prediction block for each subdivided block is given by displacing an area of the reference block. Up to sixteen motion vectors can be transmitted for each P-frame. Motion vectors are also allowed to be pointing outside the image area. In these cases, interpolation is done by duplicating the edge pixels. ¼-pel motion compensation is used for temporal prediction. 6-tap interpolation filtering is done to determine ½-pel positions. This is followed by bi-linear interpolation to get ¼-pel positions. For the more optimal profiles of H.264, 8-tap interpolation filtering is used to derive 1/8-pel positions[3].

Hadamard Transform and Quantization

Next, the prediction macroblock P is subtracted from the current macroblock to create a difference macroblock. The block is transformed using a Hadamard Transform. A Hadamard Transform is very similar to a discrete cosine transform, however uses integer values as opposed to floating-point. This fact reduces rounding errors, and eliminates mismatch values between the encoder and decoder. Also, the 4x4 size is smaller than the 8x8 size of the DCT used by prior video compression techniques, which helps reduce blocking and ringing artifacts. Its core operations can be done by only using additions and bit shifts. There is only one multiply required, and can be performed in the quantizer[5]. On a side note, the Hadamard Transform has a disadvantage to the DCT, in that the result creates a more blocky output. The advantages are the

decrease in errors because of the use of integers, and the minimization of computations. The result is then quantized to give a set of quantized transform coefficients[6].

Context-Based Adaptive Binary Arithmetic Coding (CABAC)

The coefficients are then re-ordered and entropy encoded. The entropy encoding used is called Context-Based Adaptive Coding (CABAC). [more about this] This information along with information required to decode (macroblock prediction mode, quantizer step size, motion vector information as to how the macroblock was motion-compensated, etc.) comprises the compressed bit stream[5].

Overall "Picture"

When the encoder runs, the first frame is encoded as an I-frame, or intra predicted frame. This is because a reference frame must be encoded first before more sophisticated compression can occur. Therefore, the first frame has no motion information and is encoded only using the neighboring pixels of each macroblock. From that point on, depending on the mode the encoder is running, either just P-frames will be encoded, or B- and P-frames will be alternating in order for encoding. We chose to run the encoding process without the use of B-, SP-, or SI- frames as per the baseline mode of H.264. We do, however, use other features that are not specified in the baseline mode of H.264 such as CABAC[5].

Approach

The original H.264 reference code comes as a pair of entities. As mentioned earlier the decoder entity is contained inside of the encoder entity to decode the previous frame for use to build the current frame. We concluded that if we were to implement the whole system a good place to start would be the decoder since a slow decoder would ultimately be a bottleneck of the overall system.

To start both the encoder and decoder were compiled in Visual Studio 6 and run on YUV samples to learn how the software works. YUV files were compressed and decompressed then reviewed to observe similarities. Everything ran worked fine so we decided to begin porting the software. The original reference code was built in C, however it was not a very portable application as claimed by the release information. Initial attempts to compile the decoder in other environments such as UNIX failed miserably and caused more errors than allowed by the compiler to return. Early attempts to compile the software in Metrowerks Code Warrior were also futile. It was decided at this point that the code needed to be cleaned up and made portable so that we could work with it in an environment other than Microsoft's Visual Studio 6.

Cleaning up the decoder took a great deal of time. There were libraries that were referenced that were not included and there were library references that didn't need to be referenced all over the mass of the code. There weren't even prototypes for some of the functions. Time libraries which were used to benchmark had to be taken out and the code had to be modified since there was no real time clock in the EVM. The complete code was so large that it had to be completely stored in the SDRAM and therefore it would make it slow for running. Alterations throughout the code were done to individual C files until they compiled under code composer one by one. One of the large problems we

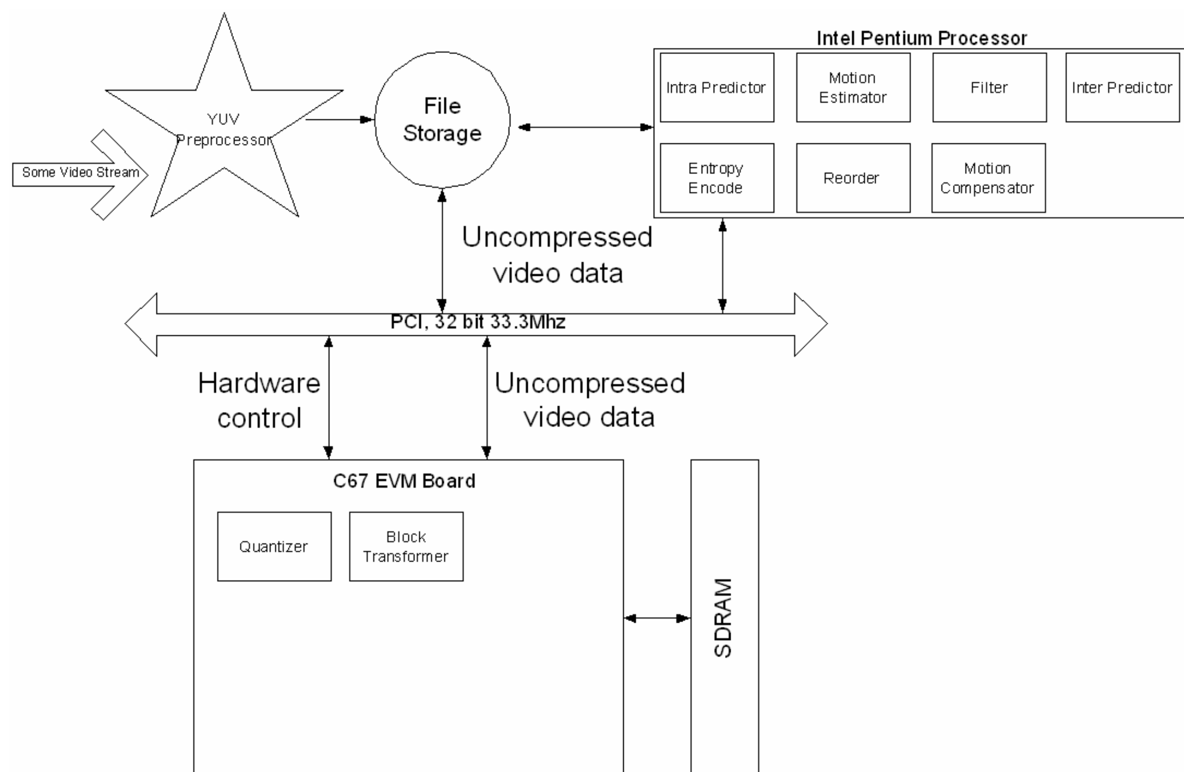
encountered while using this method came when we tried to like the program. Many global variables were defined and not declared with ‘extern’ which led to “multiply defined” errors. Also, we had to target appropriate libraries in order for the linking process to complete.

When compiling the code all together without any optimization on it took roughly 15 minutes. However, when optimizations were turned on it ran for over an hour until we decided that it wasn’t going to finish compiling in any definite amount of time and decided to cancel and take another approach. The non-optimized version of the program was moved over to the EVM and run, however it seemed to do nothing. This implemented version was built inside of Code Composer Studio and was designed to run completely on the EVM, grab files off of the hard drive, decompress them, and write them back to a file on the hard drive without the use of a proxy PC application to send and receive data. After working for roughly 4 weeks on making this version compile, link, and build to a program that could be sent to the EVM we finally had to abandon this version because working out the runtime problems would not be cost effective. The time it took to rebuild the whole program, send it to the EVM and run it with debugging information for each modification would take much longer than the time it would take to make a single modification in the code. Repeating this process until the production of a viable product was produced could not be done in any feasible amount of time. Therefore this approach was abandoned to seek a more feasible solution to our problem.

At this point we decided to attack our problem with a more bottom up approach. A stand alone EVM version was not feasible in the time frame that we had with the resources that we had available. Therefore we decided to modify our plan to build an EVM assisted version of the encoder to run on the PC. The encoder was built instead of the decoder for this implementation because we could streamline the encoder to run in a simplified “baseline” mode where it would use only the specific features of H.264 that we needed for our specific

application. A streamlined decoder would only be able to decode a similarly streamlined encoded bit stream, whereas the original decoder would be able to decode any operating mode of the encoder.

In order to determine which portion of the code we should sub task to the EVM, we stepped through the encoder to find portions of the code that the program spent the most time in. We found that a major amount of the calls were being made to parts of block.c, roughly 50,000 calls per frame. We determined that this was the portion of code that did the transform, quantization, inverse quantization, and inverse transform for luma parts of the YUV frame. This portion is used in every sub-block of every macroblock, therefore it seemed reasonable to compute this portion as fast as possible.



Final system diagram

PC and EVM intercommunications

We started with the dct_luma() function in block.c. In order to sever this portion of code from the PC side we had to step through the code to find which memory locations were being changed. Creation of two new files on the PC side 'comm.c' & 'comm.h' was necessary because within the communication library 'evm6xdll.h' TRUE and FALSE are redefined once and in 'globals.h', TRUE and FALSE are redefined a second time. Commenting out one or the other did not seem to make it work.

In order to work around this we took the pointer values of the structs in global.h and passed them into comm.c for use within comm_dct_luma(). Now that comm.c has access to the variables that is necessary for dct_luma() it acquires them from memory and stores them into an array. Communication is now opened with the EVM and a flag is set in a predefined memory location. On the EVM side 'acute.c' is waiting for the flag to be set and sends an acknowledgement to the PC that it is ready to receive data when the flag is changed. The PC is waiting for this acknowledgement and responds by sending the data array that was created in comm_dct_luma.

When the EVM receives the data, it is broken up from the single block of memory back into smaller easily more manageable variables by using DMA transfers to on-chip memory. While the data is on the on-chip memory within the EVM side it is processed in dct_luma() which was severed from the PC side. The processed values are returned using a DMA transfer to SDRAM while the EVM sends a message back to the PC that it is ready to be transmitted. It begins transmitting to the PC and sends a message back to the PC to acknowledge that it is finished.

The PC receives all the data back and stores it back into the appropriate memory locations that they came from. The PC then breaks out of comm_dct_luma() when the kill message is sent from the EVM and the program returns back to the portion of 'block.c' that was running. Inside of 'block.c' the integer is returned and the outsourced portion of code to the EVM runs transparent to the rest of the program in the same manner that it was before the EVM assistance.

Logistics

H.264 is designed to compress video in the YUV 4:2:0 format. The YUV 4:2:0 format consists of one luminance and two chrominance components. The luminance component is sampled at the 176x144 frame resolution whereas the chrominance components are down-sampled by two in the horizontal and vertical directions.

We recorded several .AVI samples using our Alaris Weecam webcam at 10 frames per second. Using an avi-to-yuv converter we found online, we converted these files to the YUV 4:2:0 format to be encoded. After a sample was encoded and then decoded, we played the output using our yuv viewer, which we also found online.

Results

Our first attempt, was to do PCI FIFO transfers for everything, which turned out to take a large portion or the total time of communications between the EVM and PC. Our initial processing times were slower than our final processing times, however, still much larger than the time required for the PC to encode by itself.

In the end, as described earlier, we used dma transfers for sending and receiving our variables between the PC and EVM. This was the method used in lab 3, and proved to be the most effective for our situation.

On the next page is a screen shot of the output given by our implementation of the encoder. Under Frame, the type of frame being processed (I or P) can be seen indicating whether intra or inter prediction is taking place.

Our original processing times for the PCI FIFO transfers:

Areas	Code Size	Incl. Average	Excl. Average
send_variables	392	8105	8030
request_transfer	116	1895	36
process_memory	472	681	396

After changing to dma transfers:

Areas	Code Size	Incl. Average	Excl. Average
send_variables	484	37646	10743
request_transfer	40	147393	12
process_memory	20	17839	6

```
Parsing Configfile encoder.cfg..................................................................
........................

------------------------------------------------------------------------------
 Input YUV file                    : newtest4.yuv
 Output H.26L bitstream            : test.26l
 Output YUV file                   : test_rec.yuv
 Output log file                   : log.dat
 Output statistics file            : stat.dat
------------------------------------------------------------------------------
 Frame    Bit/pic   QP   SnrY    SnrU    SnrV    Time(ms) Frm/Fld IntraMBs
  0(I)     22480   28 37.6271 41.8961 41.3636  66996       FRM
  1(P)      8824   28 35.5553 40.8929 41.0307  85853       FRM    24
  2(P)     18776   28 37.4533 41.1925 40.7992  90881       FRM    60
------------------------------------------------------------------------------
 Total Frames:  3 (3)
 LeakyBucketRate File does not exist; using rate calculated from avg. rate
 Number Leaky Buckets: 8
     Rmin      Bmin      Fmin
   500790     22480     22480
   625980     22480     22480
   751170     22480     22480
   876360     22480     22480
  1001550     22480     22480
  1126740     22480     22480
  1251930     22480     22480
  1377120     22480     22480
------------------------------------------------------------------------------
 Freq. for encoded bitstream       : 30
 Hadamard transform                : Used
 Image format                      : 176x144
 Error robustness                  : Off
 Search range                      : 16
 No of ref. frames used in P pred  : 1
 Total encoding time for the seq.  : 243.730 sec
 Sequence type                     : IPPP (QP: I 28, P 28)
 Entropy coding method             : CABAC
 Search range restrictions         : none
 RD-optimized mode decision        : used
 Data Partitioning Mode            : 1 partition
 Output File Format                : H.26L Bit Stream File Format
------------------ Average data all frames  -----------------------------------
 SNR Y(dB)                         : 36.88
 SNR U(dB)                         : 41.33
 SNR V(dB)                         : 41.06
 Total bits                        : 50080 (I 22480, P 27600)
 Bit rate (kbit/s)  @ 30.00 Hz     : 500.80
 Bits to avoid Startcode Emulation : 0
 Bits for parameter sets           : 160
------------------------------------------------------------------------------
```

Speed Issues

The H.264 implementation built for the PC was designed to be run with a great deal of memory at once. However, the EVM does not have a great deal of memory like a PC. The original implementation of the decoder consumed a massive amount of space on the EVM. It was roughly 4 Mbytes in size. This caused a very limited space for the data frames and any manipulation of on data. However it did fit barely on the onboard memory however it was much too large to run in on chip or even SBSRAM.

When we moved to the implementation of the encoder, dct_luma() and the respective data fit conveniently on on-chip memory. This allowed us to maximize the speed of that portion of the algorithm. Original speed issues before moving to DMA block transfers were massive. When we were using PCI transfers the time it took to process a single sub-block took much less time than was needed to transfer the memory. The memory transfer of all the variables on and off of the EVM from the PC was definitely our bottleneck in processing dct_luma() on the EVM.

Obstacles to Overcome

Once again, the reference code provided by Karsten Suehring and the Joint Video Team was poorly written and commented. H.264 codec was designed with the video compression layer and the network abstraction layer. Flexible Macroblock Ordering (FMO) allows the blocks to be sent in an order different from raster scan order. This makes the algorithm faster, however necessitates a need for wrappers to be placed around each block to ensure that the decoder performs in the correct order.

Large data structures and objects are used by the algorithm, and there was a problem that was fixed about how to go about dealing with these large structures as well as pointers to these structures. We got around that problem by creating an array of all the variables that were needed and that were to be changed by the dct_luma function. Thus, entire structures did not need to be sent, rather only portions of the entire structure. This saved processing time as well as memory greatly.

Further Work

Our project could be improved upon and extended in the following ways. More functions could have been ported onto the EVM that were similar to the one we did, i.e. doing transform and quantization for the chrominance blocks, as opposed to only doing it for the luminance blocks. If we had ported a similar function onto the EVM that used similar variables, this definitely would have cut back on the time to send variables back and forth from the PC to EVM and thus, cut back on total encoding time. Thus, stepping back another level and implementing more of the algorithm would be the next step for this project. Given the fact that we were working on getting the decoder to work on the EVM for too long, we were not able to do this.

References

[1] Ngan, King N., et al. "*Advanced Video Coding: Principles and Techniques*." Elsevier: Amsterdam. Volume 7, 1999.

- Information on SAD
- Motion Vector References

[2] Nee, Kurt and Roesle, Chad. "Feasibility of Implementing an H.263+ Decoder on a TMS320C6X Digital Signal Processor". May, 1999

- Similar final project done at University of Texas using earlier codec

[3] Schäfer, Ralf; Wiegand, Thomas; Schwarz, Heiko. The Emerging H.264 /AVC Standard, *EBU Technical Review*, Jan. 2003.

- High level technical overview and standard comparison

[4] Sühring, Karsten Software available: http://bs.hhi.de/~suehring/tml/ Provided by, last updated 02/03/2003

- encoding and decoding software in C

[5]Vcodex: H.264 tutorial white papers at: http://www.vcodex.fsnet.co.uk/h264.html

- Breakdown of major components (Inter/Intra prediction module, CABAC)

[6] Wiegand, Thomas. "Study of Final Committee Draft of Joint Video Specification (ITU-T Rec. H.264 | ISO/IEC 14496-10 AVC)." December 2002.

- Official H.264 specifications