Hardware Software Codesign

Acceleration of HDMI video processing on Zynq (Part 1)

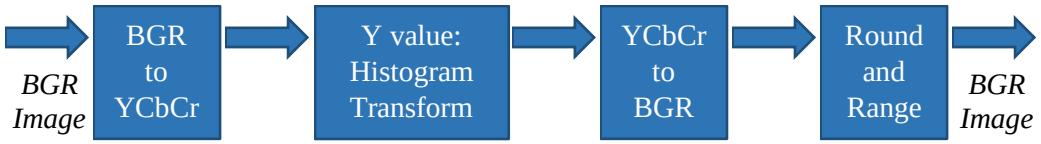
(Author: Lukáš Kekely, ikekely@fit.vutbr.cz)

Project part 1

3. Examine the source code of the provided video processing implementation for general CPU.
4. Port the sample implementation onto the PYNQ-Z2 board to process 720p HD HDMI video using only embedded ARM cores.
5. Profile your PS implementation to measure the achievable performance and identify critical parts of the processing that require further hardware acceleration.
6. Summarize part 1 outputs and upload them into the information system.

3. Sample implementation

The sample contrast enhancement implementation is provided in the *contrast_enhance.py* file from the downloaded ZIP archive. The implementation uses histogram equalization (HE) approach based on the weighted and thresholded histogram equalization (WTHE) method. The core idea is to evenly distribute a narrow range of intensity values from the original image to the total available spectrum. In other words, based on the original image histogram, a value transformation is performed that will result in an image with a more evenly spread-out histogram. The standard HE methods can process only one-dimensional data (intensity values). To enhance the contrast of colored images represented with three color dimensions (Blue, Green, Red), conversion to another color space that uses luminance (L) or grey (Y) channel is needed. The L or Y channel can then be used as intensity values in HE to transform the image without significant changes to the hue and saturation of colors. The sample implementation uses YCbCr color space for this purpose.



The implemented image processing, as illustrated above, consists of the following primary steps:

- convert color space from default BGR representation to YCrCb (**_convert_BGR2YCrCb*),
- compute a histogram of Y values and transform/equalize them (*process_value** in *WTHE*),
- convert color space from YCrCb (with new Y) back to BGR (**_convert_YCrCb2BGR*),
- round and threshold any underflow or overflow in BGR values (functions **_ float2byte*).

Note that for every step, there are two versions of corresponding functions. One uses native pixel-by-pixel processing (function *processing_native*), and the other uses vectorized *numpy* operations over whole frames (function *processing_numpy*). The results of both versions are the same, but the vectorized one is considerably faster in Python. On the other hand, the native version demonstrates the functionality more clearly to better understand and find performance bottlenecks during profiling.

The function *transform_update* from the *WTHE* class implements the transformation table (T) computation based on image histogram (H). This update function is taken out of the pipeline to enable processing with only one pass over each frame and no delay between frames. Specifically, two copies of H and T are used. The first (active) copy updates H, and its T is applied. Meanwhile, the other copy uses H to obtain a new T in the separate thread. After N frames (N=4), the processing is switched between the two copies. As a result, each block of N frames is equalized with T based on H from N frames ago as illustrated below. H from frames 0-3 is used to obtain T during frames 4-7, while H[4-7] is computed on the second copy. This T[0-3] is then applied on frames 8-11 while T[4-7] is calculated on the second copy and so on. Implementation of this mechanism can be seen in the *main* function. The rest of the *main* function implements command-line option parsing, video initialization and cleanup.

	Frames	0-3	4-7	8-11	12-15	16-19	20-23
1st copy	**State**	Active	update	Active	update	Active	update
	T	empty	T[0-3] →	apply	T[8-11] →	apply	T[12-15] →
	H	H[0-3]	clear	H[8-11]	clear	H[16-19]	clear
2nd copy	**State**	update	Active	update	Active	update	Active
	T	empty	empty	T[4-7] →	apply	T[12-15] →	apply
	H	empty	H[4-7]	clear	H[12-15]	clear	H[20-23]

Finally, the program can be executed by calling *python3 contrast_enhance.py*. All parameters and their description can be seen using option -h. Input and output can be specified by -i and -o options. Options -c and -l control the length of execution. Pressing Ctrl+C can terminate the program prematurely.

4. Porting to PYNQ-Z2 board

Start by learning more about how PS-based video processing and HDMI drivers work on the PYNQ board. Use provided Jupyter Notebooks accessible through a WEB browser. Connect to your PYNQ board WEB page and navigate to examples in *video* folder. Go through the HDMI introduction and HDMI video pipeline examples. For additional information, use the PYNQ video part of the official documentation. Familiarize yourself especially with correct procedures to:

1. Load and initialize overlay (bitstream) to the FPGA programmable logic (PL). The existing base overlay already includes HDMI input and output pipelines with appropriate controllers of the physical HDMI ports. The pipelines are connected to video DMA module to transfer image data between PL and main DDR memory accessible from video drivers on PS.
2. Configure and start the input and output video drivers to enable video transfers over HDMI. The video drivers are automatically instantiated during overlay object initialization and are accessible as its *video.hdmi_in* and *video.hdmi_out* attributes.
3. Read and write frames to receive and stream video over HDMI ports.
4. Safe cleanup and closing of HDMI drivers at the end of processing.

At the beginning of the contrast enhancement code porting, create a copy of the *contrast_enhance.py* file and name it *video_ps.py*. This will serve as a template for PYNQ specific implementation. Now change appropriate parts of the code to replace default OS camera capture with HDMI input port and on-screen playback with HDMI output port. The -i and -o options and possibility of file input/output should be retained, change only behaviors when these options are not used. Follow these steps:

1. Add base overlay initialization by constructing object of class *BaseOverlay*.
2. Replace default camera input initialization with HDMI input port *config* and *start*. Get video parameters from *mode* attribute of HDMI input (*VideoMode* object).
3. Add HDMI output *config* and *start*. Create and use *VideoMode* object with constant parameters for resolution 1280x720, 24-bit BGR pixel format and 60 FPS.
4. Replace video frame reading from camera with reading from HDMI input (method *readframe*).
5. Replace on-screen playback with writing to HDMI output (method *writeframe*). Note that frame format can changed during processing. To reestablish HDMI compatible format use *newframe* method to get empty frame, copy the data (*o[:] = i*), and write the reformatted frame.
6. Do not forget to do the correct cleanup and *close*. Ending without proper HDMI driver cleanup could result in problems with consequent runs and would require PYNQ reboot.

To test the video enhancement implementation on the PYNQ board, follow these steps:

1. Create a custom folder on the PYNQ-Z2 board, for example folder *hsc*.
2. Move the *video_ps.py* file and *data* folder to this folder on the board.
3. Test the HDMI video functionality on the PYNQ board by calling *python3 video_ps.py*:
   - You must be root for the script to load the PL overlay. Use *su* command and *xilinx* password.
   - Use -i and -o options to read and write local video files on the SD card.
   - Omit the -o option to stream video to the HDMI output port. You can use an HDMI cable to connect a monitor to PYNQ's output HDMI port to see the video playback.
   - Omit the -i option to read video from the HDMI input port. Live HDMI cable must be connected to the PYNQ's input HDMI port. Use HDMI camera or computer as video source.

The same video processing functionality should be achieved, including printing the same command-line messages. Sample video and image files from *data* folder can be used for testing. Note that every version of Python and/or OS installation can have variations in video codecs. Slight discrepancies in the processed values between different platforms are therefore possible.

5. Performance measurement and profiling

Measure the achievable performance of video processing on the PYNQ's ARM-based PS using the implementation created in the previous chapter. Basic information should be printed at the output of your *video_ps.py* script. How fast can the script process HD video (1280 x 720 pixels)? Is it sufficient for real-time playback and enhancement? Note that read and write operations on the SD card can negatively affect the achieved performance. Therefore, for the most accurate measurements connect an HDMI source (e.g. notebook or computer set to 1280 x 720 resolution) to the PYNQ board's HDMI input port and do not use -i nor -o options.

Analyze the implementation and identify the critical parts of the video processing pipeline. Select a subset of functionality for hardware acceleration in PL to achieve higher FPS rates, ideally up to 60 FPS. Standard Python profiling tools can aid you in the analysis. The modules *profile* or *cProfile* can be used to obtain a set of statistics that describes how often and for how long various parts of the program are executed. To get them, use function *run* or construct a profiler object and use *enable* and *disable* methods. The profiling statistics can be then formatted into reports via the *pstats* module. Use class *Stats* and method *print_stats* to print the results. For a clearer view of the implementation's bottlenecks, use method *sort_stats* with various sort keys. Cumulative time spent in a function from invocation till exit may be interesting here. Alternatively, the total number of function calls can be viewed for native pixel-by-pixel implementation. Note which parts of the functionality are executed on each pixel of each frame and which functions are called less often.

Write down the achieved FPS measurements (with disabled profiling) and store the profiling statistics. You will need them for report creation at the end. Decide where would be the ideal place to split the video processing between PL and PS? Which functions should be accelerated in the FPGA?

6. Required project outputs from part 1

The following is required:

- File ***video_ps.py*** with PS only implementation of the specified video processing algorithm.
  - Do not forget to disable profiling by default. Comment it out or hide it behind option -p.
- A technical report named ***report.pdf*** that consists of:
  - A short description of steps performed during code porting to PYNQ's PS.
  - A brief result of PS performance measurements and profiling.
  - Description of how you would divide the processing between PL and PS and why.
  - The technical report should not exceed one page in A4 format.

Pack all the mentioned source codes and the technical report into a zip archive named ***project.zip*** and upload it into the information system no later than the specified date.