Video processing application using EMB²

This simple application makes basic processing on a video file by exploiting the possibilities offered by EMB². In order to do it uses one of the fundamental building blocks of the library, i.e. *Dataflow*. Dataflow provides a simple way to model pipelines or, more generally, situations in which a stream of data must be analyzed by some processing nodes that can be organized in a network.

Handling the IO

For handling the IO of the program we use the ffmpeg C API. Instead of calling directly the ffmpeg functions, the application includes some wrapper classes with a very simple interface, such that in a future tutorial the input/output part would not distract from the actual purpose of the tutorial (parallelization using EMB²). These classes are:

- *InputVideoHandler:* as the name suggests, this is the class responsible for reading from the video. The class can only be instantiated by providing a name for the input file as *char** (default constructor is not accessible). The main method of the class is *bool readFrame(AVFrame* frame, int* success),* which is essentially a wrapper for the *av_read_frame* method of ffmpeg. It reads either the first frame or the next frame of the input and stores it in the position pointed to by the first parameter (*AVFrame* is the struct used by ffmpeg for packing frame information). The second parameter, *success*, is a code that tells us if the pointer *frame* points to a valid frame; this is necessary because one call of the *readFrame* could return only a partial frame. The method returns true if there are still frames to process, false if we reached the end of the file.
- *OutputVideoBuilder*: in a symmetric way to *InputVideoHandler* this class handles the writing-to-file part. It can only be instantiated by providing an output name. The constructor requires also a second parameter, which is a pointer to an *AVCodecContext.* It is sufficient to know that this struct contains information about the way the encoding (compression) of the frames should be done. In the application, we will take this information directly from the input handler. The two main methods here are *void writeFrame(AV* frame)* and *void writeVideo()*. The former writes the provided frames to output, while the latter must be called after the last frame has been written and executes the finalization of the file.
- *FrameFormatConverter:* it is very likely that the frame we will extract using the *av_read_frame* function inside the *readFrame* method will not be in RGB format. This depends on the codec that corresponds to the specific video format we are using (for .mpg, for example, the frame will use the YUV color space). Since it seemed more natural to work with pictures in RGB format, because it is perhaps more intuitive and well known, this class provides an easy way to converting frames from the original format to RGB. Note that after the frame has been processed it will be needed to convert it back to the original format, because we want to obtain a video with the same extension of the original one. The main method is *void ConvertFormat(AVFrame** input, AVFrame** output, ConvertionType ct),* where ConvertionType is a enumeration type that can take the values TO_RGB or TO_ORIGINAL.

Finally, note that for simplicity the audio tracks of the original file (if present) will not be written to output.

Structure of the program

The structure of the program is straightforward. Thanks to the *InputVideoHandler* one frame after the other is read from the input file. The picture is then converted to RGB, modified (e.g. by applying some simple filters), converted to the original format and then written to the output file.

Where is there space for parallelization in such an application? There are actually two levels of parallelization possible:

1. If we plan to work with each frame as an individual (without caring, for example, for what comes before and what comes next) the processing can be executed in parallel because they do not depend on each other.
2. Each picture could be divided into parts which could be analyzed in parallel.

In our program we limit ourselves to the first level of parallelization. The reason for this is twofold. First of all, we should be able to get a substantial speedup (maybe close to linear in the number of cores, if we neglect the IO part) even when we allow ourselves to work on many frames in parallel. This because a video file is composed by a large number of frames and we can expect all the cores to be exploited during the duration of the program. The second reason is that the most expensive filters are the ones that work on a neighbor of a pixel (and not on the pixel only). This does not cause any problem inside the regions we could choose to divide the frame in. On the borders, however, different processes could try to access the same pixels at the same time causing races conditions. Obviously this is a problem that can be solved (for example by treating the boundary pixels separately), but it is maybe more difficult to achieve a good speedup in this situation. In any case, combining the two levels or parallelization would not be too beneficial, because as said before a video files contains a number of frames considerably higher than the number of cores we can use.

We could try to model our program as a network. In this scenario, we could imagine a graph composed by 5 nodes. One node (a “source” node, using the vocabulary of EMB²) is used to read from file. Two nodes are used for converting formats and another processing node that applies filters to the frames. This node can process in parallel different frames, because as said before frames are assumed to be independent. Finally, a “sink” node is used to write to a file. A scheme of the network is sketched in Figure 1.

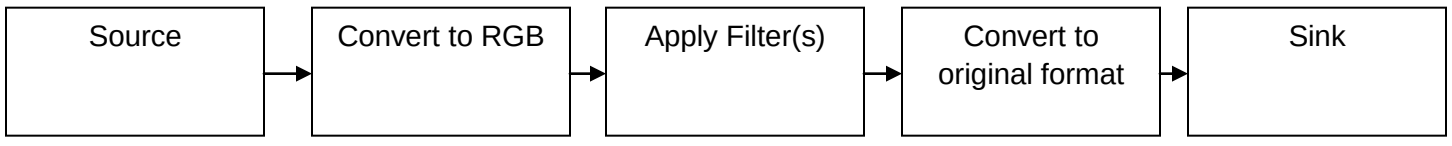

Figure 1.

Filters

Filters are implemented in the filters.h header file.

- applyShuffleColor: shuffle the rgb value of the picture. Red is assigned to green, green is assigned to blue, blue is assigned to red
- applyBlackAndWhite: convert the frame to black and white by setting the rgb values to the average of the three.
- applyNegative: convert the frame to its negative by applying the rule “color = 255 – color” to each channel
- edgeDetection: apply sobel filter to frames
- applyCartoonify: apply edgeDetection (edges will be colored in black) and then color reduction to obtain a cartoonish stile. The filter accepts two parameters: threshold, which determines how tick the edges will be, and discr, which determines the range of colors to use for the color reduction.
- changeSaturation: change saturation according to an amount passed as parameter.
- applyMeanFilter: apply mean filter to a frame. The size of the kernel must be specified as method argument.
- applyVariableMeanFilter: apply variable blur to a frame. The image is divided into (fixed) horizontal stripes, and mean filters with different kernels is applied to each one of those.