Video UnScrambler

In the 1980s and 1990s, the processing capabilities that could be embedded in a TV decoder were limited compared to what we can do today. Processing was still mainly carried out using analog circuits for everything related to frame generation and synchronization signals. A minimal digital component managed the transmission/decoding of encryption keys and pay-TV subscription information, embedding them in the analog signal during dead air time. As in the early days of radio broadcasting, the user side had to be simple and inexpensive in order to democratize its use as quickly as possible. This required simple but effective encryption that could be decrypted with little computing power. Several versions were marketed depending on the year, country, and broadcaster, but the general principle was always to mix the image lines. Sometimes the lines were even split in two and/or mixed on the two interlaced fields (interlacing disappeared with digital broadcasting). Processing capacity was so limited that the lines were mixed in packets of only 32.

Video scrambling Principle

For a video of dimensions (height x width), we will simply apply the principle of line scrambling, frame by frame, without limiting the number of lines:

A line with index **idLine** (line 0 being at the very top) in the clear input image will be sent to position **((r + (2s+1)idLine)%size)** where

r is an offset encoded on 8 bits
s is a step encoded on 7 bits
the pair **(r,s)** forms the **encryption key**.

The encryption here is symmetric with private key **(r,s)**.

This principle only applies if **(size)** is a power of 2, which is rarely the case.

To adapt to all video heights **(height)**, we will apply the line permutation iteratively:

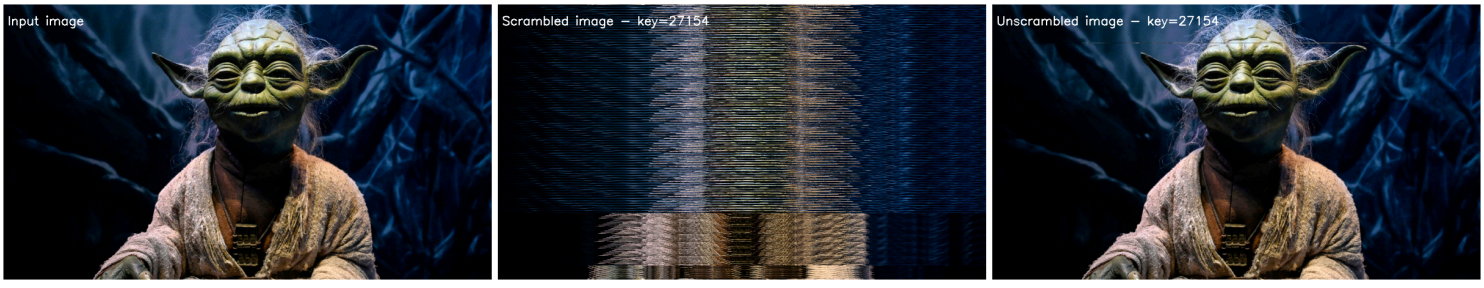
iteration #1: we search for $(\mathbf{2^n})$ as the largest power of 2 less than or equal to **(height)** and apply line scrambling from 0 to $\mathbf{2^n - 1}$.
iteration #2: we proceed with the untreated lines: from $(\mathbf{2^n})$ to **(height-1)**.
iteration #3: and so on until the last line **(height-1)** is processed. It may potentially remain unchanged.

In the end, all lines have been processed and moved (except possibly the last one), but since we proceeded from top to bottom with increasingly smaller blocks, this means that the further down you look in the encrypted image, the less *unreadable* it will be.

The closer **(height)** is to $\mathbf{2^n}$, the fewer iterations there will be and the more unreadable the encrypted image will be, because the first residue itself will be small in size.

Example of encryption/decryption:

Image 675 x 1200.
4 iterations of sizes 512, 128, 32, and 2 (the 3 big blocks are clearly visible)
Scramble + unscramble: 2ms.



Cracking the key: decrypting the video without knowing the key

When the key is available, decrypting the video frame by frame is child's play: you simply invert the line permutation. With enough computing power, this can even be done in real time.

However, when the key is not available, it must be recovered. For this project, you will implement a brute force search for the key that was used to encrypt a given image. To do this, you will try all possible keys $(2^{15} = 32768)$ to decrypt the encrypted image, then evaluate the quality of the result using a selection criterion. The key that yields the best result will be considered the correct one (fingers crossed).

Moreover, in our case, chances are high that you will be able to tell, with the naked eye, whether the decrypted image is correct or not.

In an unencrypted image, consecutive lines often look very similar, since they represent areas that are very close in the scene.

The idea is therefore to measure the similarity between each pair of consecutive lines in the decrypted image with the tested key, and then sum these values to obtain a global score.

The better the score, the more likely the image is correct.

To measure the similarity between two lines of the image, several approaches are possible. Our first proposal is to use the simple Euclidean distance between two lines.

Let x and y be two lines of the image, each encoded as a sequence of n pixels (32-bit integers in GL) with values denoted x_i and y_i at column i.

Selection Criterion #1: Euclidean distance

It is defined by the following formula, where the 1/2 exponent is equivalent to the square root (avoids a display bug in some browsers):

$$d(\mathbf{x},\mathbf{y}) = \left(\sum_{i=0}^{n-1}(x_i - y_i)^2\right)^{1/2}$$

> The Euclidean criterion is sensitive to lighting variations. To be more robust, it may be useful to normalize the gray levels of the lines before computing the Euclidean distance. This can be done by subtracting the mean gray level of each line before computing the Euclidean distance.
>
> Moreover, to avoid integer overflow when computing the Euclidean distance, one can either perform the calculations in floating point (double) rather than integer, or divide the criterion by the sum of the variances of the two lines (a kind of normalization). This is the idea behind Pearson correlation, proposed below.

Selection Criterion #2: Pearson correlation

It is defined by the following formula:

$$\rho(\mathbf{x},\mathbf{y}) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}; \sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{x}$ and $\bar{y}$ are the means of the pixel values of lines x and y.

The Pearson correlation coefficient ranges from -1 to 1. A value close to 1 indicates strong positive correlation (the two lines are similar), a value close to -1 indicates strong negative correlation (the two lines are opposite), and a value close to 0 indicates no correlation (the two lines are independent).

Objective

The goal of this assignment is to implement a video unscrambler that can decode a scrambled video using a Jetson Nano. The video stream will be provided as a file. The unscrambled video should be displayed on the screen and saved to a file. The unscrambling process should be done using CUDA to leverage the GPU capabilities of the Jetson Nano.

> **Note**
>
> The audio tracks will be ignored for this assignment. Though the audio encryption process was entirely analog by that time, a digital GPU version has been proposed in a past assignment (**Crypto WAV: Lab asssignment**), but it is not required here.

0:00 / 0:28

The video above is provided as an example. It has been encrypted according to the principle described here. You can download it by following **this link** . The C++/OpenCV program used to perform the encryption is also available by following **this link**. Another copy is available in the directory */home/jetson/Documents/NANOCODES/videoScramble/* on each Jetson Nano.

You can use it to encrypt other videos of your choice, or to verify that your decryption program works correctly (You'll need to write the reverse permutation function). Depending on where you plan to use the scrambler, it can be compiled/run as follows:

On the Jetson Nano

open a terminal and navigate to the directory */home/jetson/Documents/NANOCODES/videoScramble/* where the file *videoFileProcessor.cpp* is located.

Compile the program with the following command:

```
g++ -o VideoFileProcessor videoFileProcessor.cpp -O2 -Wall `pkg-config --cflags --libs opencv4` -std=c++17
```

Run the program with the following command:

```
./VideoFileProcessor inputFile_path outputFile.avi encryption_key
```

On your own computer

Opencv v4 is required. If you don't have it yet, please follow the instructions on the official OpenCV site.

Create a CMakeLists.txt file in the same directory as videoFileProcessor.cpp with the following content:

```
cmake_minimum_required(VERSION 3.10)
project(VideoFileProcessor)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(OpenCV REQUIRED)

add_executable(VideoFileProcessor videoFileProcessor.cpp)

target_link_libraries(VideoFileProcessor ${OpenCV_LIBS})
```

Then run the following commands in the terminal:

```
mkdir build ; cd build
cmake .. -DOpenCV_DIR=/usr/local/include/opencv4 # Adjust the path if necessary
make
./VideoFileProcessor ../inputFile_path outputFile.avi encryption_key
```

Note

The above program is just a dirty example to illustrate the encryption principle and to allow you to encrypt every video samples you'd want to. It is not optimized for performance, especially for real-time processing.

The display of both videos (scrambled and unscrambled) will probably be choppy on most of your computers (not to mention the Jetson Nano). The focus here is on the unscrambling process, not on video playback.

Don’t hesitate to improve it if you want to, but it’s not mandatory for this assignment. Don’t forget to share your improvements with the class (and the prof. :)) if you do so.

Project Details

You will implement a C++/CUDA program that performs the following tasks:

1. **Video Input**: Read a scrambled video file (e.g., *IHear_scrambled.m4v*) using OpenCV.
2. **Key Search**: Implement a brute-force search to recover the correct encryption key (r, s) by trying all possible combinations and evaluating the quality of the decrypted image using the Euclidean distance or Pearson correlation criterion.
3. **Video Unscrambling**: Once the correct key is found, unscramble the video frames using the selected key.
4. **Video Output**: Display the unscrambled video in a window and save it to a new video file.
5. **Performance Optimization**: Ensure that the unscrambling process is optimized for real-time performance using CUDA.

The main program file already exists on each Jetson Nano device and is named *video-viewer.cpp* and can be located in the following directory:

```
/home/jetson/Documents/NANOCODES/cudavideoproc/
```

In his great kindness, the professor provides you with the main video playback loop and a sample dummy kernel. This will allow you to focus on the GPU part of the code you need to write.

Implementation Notes

Optimization: Since the image processing will run on the Jetson Nano GPU, ensure that your kernels are optimized for GPU processing using **CUDA**.

Real-time computation: it is strongly encouraged to display the actual frame-per-second rate at runtime.

Deliverables

1. **Workgroups** You will work by pair of students.
2. **Source Code**: Implemented in C++/CUDA, with appropriate comments for readability. CUDA kernels and C++ function may be defined in one single file (.cu), even if it’s not the best way to organize a projet in the real life.
3. **Demonstration/presentation**: A brief presentation explaining the CUDA design and speed optimization (10 minutes max) along with a live demonstration of your program using the provided video file and showing the calculation in real-time.

Grading Criteria

Your project will be evaluated based on the following criteria:

Correctness: The program correctly unscrambles the video using the brute-force key search method and displays/saves the unscrambled video.

Performance: The program runs efficiently on the Jetson Nano, leveraging CUDA for real-time processing. The more real-time, the better.

Code Quality: The code is well-organized, commented, and follows best practices for C++ and CUDA programming.

Presentation: The presentation is clear, concise, and effectively communicates the design and implementation of the project.

Good luck, and remember to leverage GPU capabilities **effectively** !

Warning

Deadline

You are invited to send me your source code via e-mail as an attachment. The subject must be [M1IoT videoUnscramble].

I must receive your codes by Wednesday, December 10th. Any code that would not be sent by this date will receive a grade 0/20.

On the 12th of december, you will present your design during 10 minutes and then answer a few questions for 5 more minutes.