SPM Project - Video motion detector

Geremia Pompei (MAT. 638432)

July 12, 2022

1 Introduction

The aim of this project is detect movements inside a video located in the file system. This is done comparing pixels of a background with the current frame of the video. Before the comparison frames are transformed applying them a grayscale and smoothing transformation. Given in input a parameter k the program is able to count the number of frames that have $k\%$ different pixels from background image.

2 Process detail

2.1 Sequential implementation

At the beginning to solve this task it's decided to scroll frames of a video in a sequential way. Before the scrolling it's processed and setted as background the first frame while then this is changed if the current frame is detected. During the scrolling is applied the sequential process showed on Figure 1. The *Detector* is an object that is able to transform an image and understand if this is similar or not with respect a background frame.

2.1.1 GrayScale

Method of detector that takes a colored image and transforms this in its black and white version. This is done scrolling the matrix of a frame by rows and

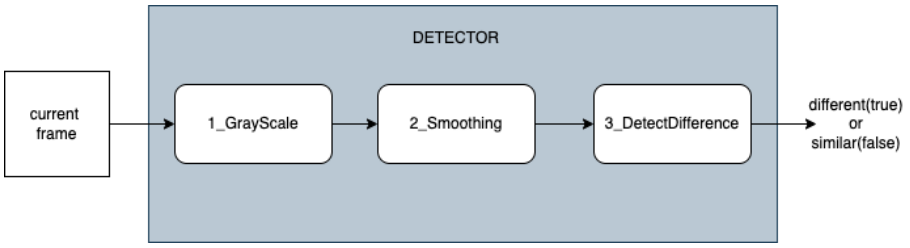

Figure 1: Sequential process

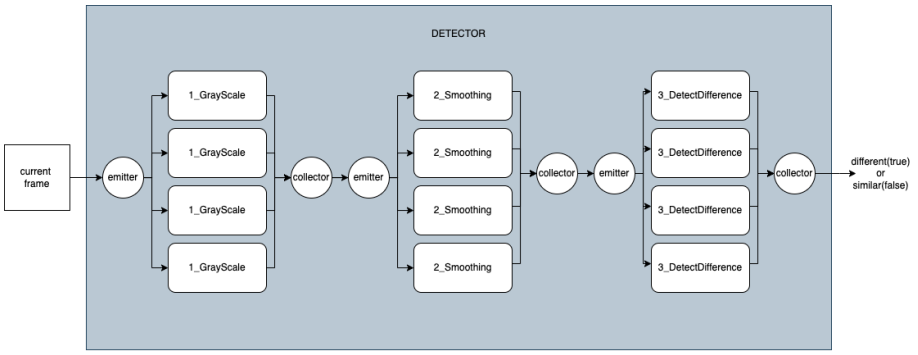

Figure 2: Data parallel process

columns and for each pixel computing the average of color channels. The result of the average is setted as value for each color channel of the same pixel.

2.1.2 Smoothing

Smoothing method takes the frame transformed by *GrayScale* and convolves this with a kernel. The kernel used in this project is a 3x3 matrix of $\frac{1}{9}$ values so an average kernel. For the convolution is scrolled the entire image by rows and columns and for each pixel is multiplied the current and the neighbours pixels by the kernel. The results of these multiplications are summed and putted inside the color channels of the current pixel. The image that is transformed here has a blur effect that change in intensity with respect the size and the type of filter. Applying this kind of average filter the effect is soft. Before applying the smoothing is cloned the original frame and is used this during the computation to transform the original one. It's used the clone frame to avoid to change neighbour pixels during smoothing process.

2.1.3 DetectDifference

Having a background frame this method is able to understand if a given smoothed frame is similar or not. Scrolling together the smoothed current frame and the background one by rows and columns it's counted the number of different pixels. If this number is bigger than the given threshold k the method return true because the current frame is detected as different otherwise return false.

2.2 Parallel implementation

2.2.1 Data parallel process

Given the sequential implementation the first thing done to parallelize it is apply *data parallel patterns* to the three methods:

- GrayScale → MAP: because this is a transformation pixel by pixel

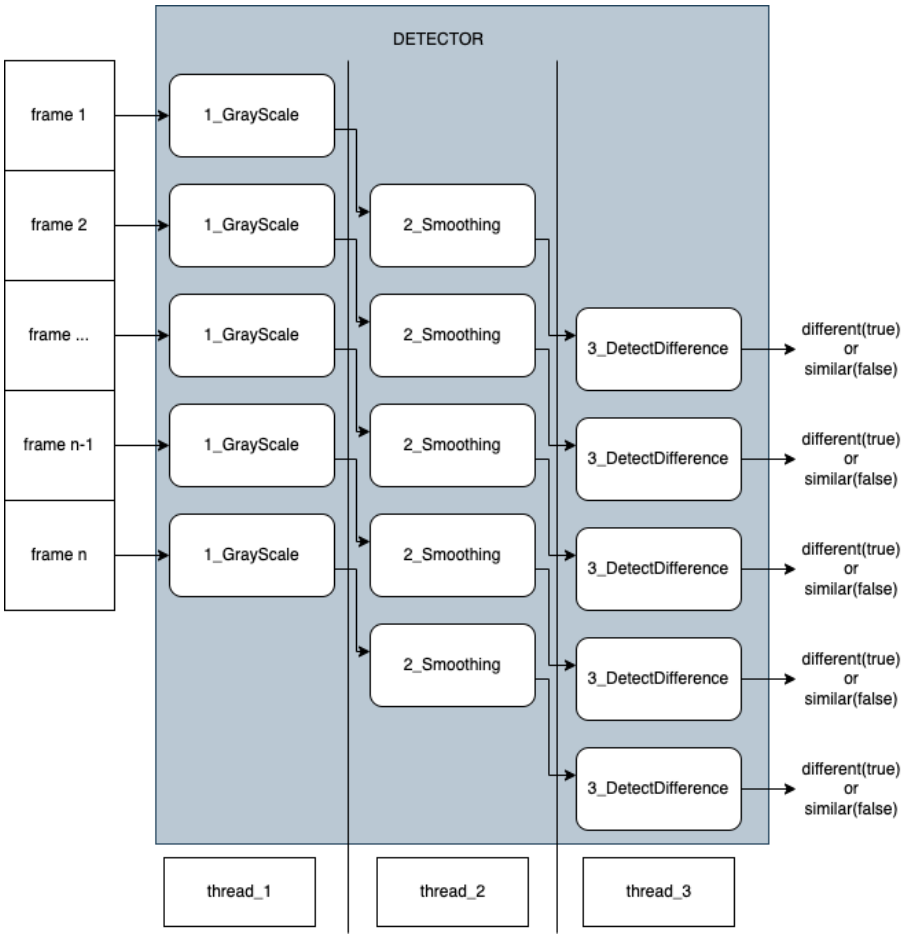

Figure 3: Stream parallel process

- Smoothing → STENCIL: because to smooth an image it's applied a convolution that uses current pixel and neighbours
- DetectDifference → REDUCE: because this take in input a frame and produces a unique result

As shown in Figure 2 each methods are treated in similar way. It's decided to wrap the particular behavior of each pattern inside a function and solve each of them passing this function to a unique way to parallelize it using an emitter, a thread on different chunks and a collector.

2.2.2 Stream parallel process

A further way to optimize the project is to operate on the streaming of frames. These matrices are extracted in a sequential way from a source. There is a lag between the extraction of a frame and the following given by the transformation applied on them. To fill the time gap it's implemented a *pipeline* with respect the the transformation functions and the detection. As shown in Figure 3 are used three threads each of them for one of the three functions. The main thread is able to extract the frames one after another and pass them in ordered way to the first function on the first thread and so on.

2.3 Brief view on code structure

To approach the project in a modular way it's decided to use two main abstract classes:

- **Detector** is able to transform images with *grayscale* and *smoothing* transformations and using *detectDifference* method to classify current frame as similar or not.
- **FramesShifter** is able to retrieve frames from video source and use *Detector* on them. It is also able to count different frames and replace background frame with the current when this is detected.

Each of these abstract classes has three children:

- *Sequential* child
- *Parallel* child with *native thread* implementation
- *Parallel* child with *fastflow* implementation

Using this structure is possible test combined parallelization to understand the better configuration.

3 Experiments

Experiments are done on video of one minute and threshold setted to 0.4. At the beginning it's shown the comparison of speedup and efficency reached by native thread and fastflow implementaion with respect ideal speedup and efficency. This is represented at Figure 4. Here the pipeline is not used, in fact the stream of frames is sequential with respect transformations.

In Figure 5 is shown the speedup and the efficency given before adding a native thread implementation of pipeline on streaming of frames.

Figure 6 is the same speedup and efficency of before using pipeline but with fastflow implementation.

Figure 7 shows time related to each single transformation functions parallelized using data parallel pattern with sequential stream.

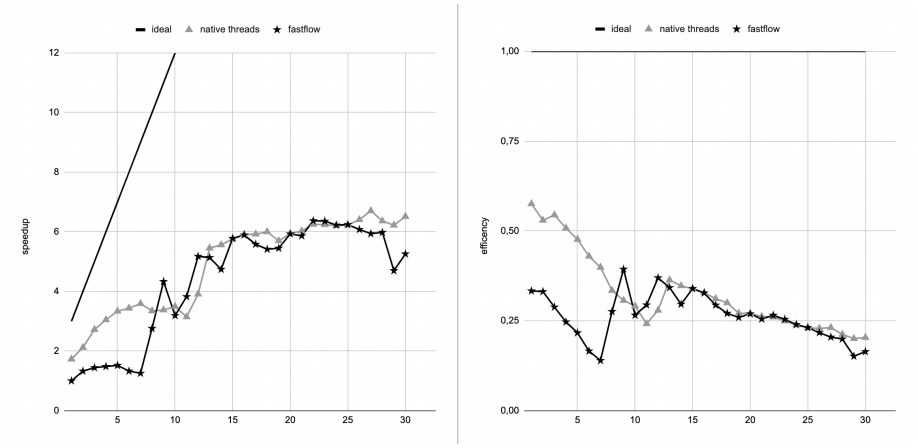

Figure 4: Speedup and efficency of native thread and fastflow parallelization with respect ideal one using sequential stream of frames

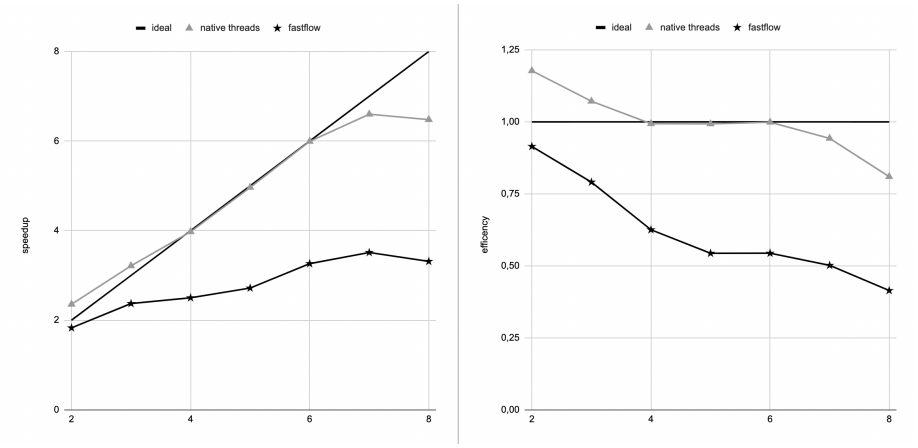

Figure 5: Speedup and efficency of native thread and fastflow parallelization with respect ideal one using native thread pipeline

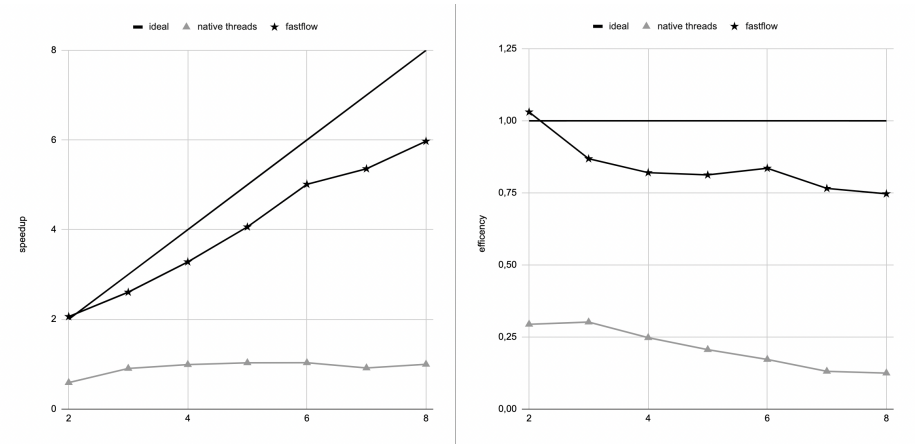

Figure 6: Speedup and efficency of native thread and fastflow parallelization with respect ideal one using fastflow pipeline

NW	1_GRAYSCALE	2_SMOOTHING	3_DETECT_DIFFEREN	TOTAL_TIME
SEQUENTIAL				
1	58,5232	155,079	33,1423	254,923
NATIVE_THREAD				
2	42,0765	107,901	23,6533	179,917
3	34,8335	88,1836	18,2255	147,581
4	28,9449	70,0165	14,5455	120,304
5	20,2419	54,7517	11,8554	93,6184
6	18,8897	48,2912	9,72436	83,6099
7	16,345	43,8731	8,09738	76,3847
8	16,2264	42,6791	8,41845	74,1416
FASTFLOW				
2	55,3149	143,973	32,4901	237,193
3	56,878	130,313	44,2888	255,028
4	46,6213	105,42	33,7105	192,315
5	42,164	93,6062	33,9748	176,833
6	42,0598	86,7993	34,1887	171,978
7	42,2034	82,9638	35,8556	167,973
8	50,8081	90,2581	44,6335	192,093

Figure 7: Time related to different transformation functions optimized

4 Analysis

As seen in experiments using parallelization there is an optimization with respect sequential case. Of course this acceleration is not equal to the ideal cases for the overhead.

In first parallelization approach (Figure 4) that parallelizes only the transformation functions the native thread and fastflow implementation reach similar results that are not close to the ideal case. The high overhead in this case is probably given by the division in chunks in native thread implementation. The chunk technique brings to have some threads that finished their computation and other not.

Figure 5 shows optimization of first approach using the pipeline developed with native threads. Using this stream parallel pattern the speedup increases. The combination of pipeline and transformation functions optimization both implemented with native threads brings the speedup to be very similar to the ideal. Fastflow implementation of trnsformation functions optimization instead reach worse results.

The behavior is inverted in Figure 6 where is used the pipeline implemented with fastflow. In this case in fact the combination with data parallel optimization developed with fastflow reachs good results slightly worse than ideal. Here native thread optimization using data parallel patterns combined with fastflow pipeline reaches very bad results.

Figure 7 shows data related to the time spent on different transformation functions run in sequential, native thread and fastflow implementation. The smoothng transformation results to be very heavy in term of computation with respect others. This behavior is replicated also using pipeline implementations. Another optimization to do could be using farm parallel pattern on this particular transformation function optimizing pipeline. Doing this can be avoided the smoothing bottleneck.

5 Instructions

There are two script files in the main folder of project to compile and run the code on remote machine. To build the program there is the `./build.sh` file that is able to construct the executable of the code in `./src` directory. There is also a `./test.sh` file to run a simulation of the program with some alternative configurations. The basic commands to do this without script files are

```
g++-10 src/main.cpp -o3 -o build/main ‘pkg-config --cflags
 opencv4‘ ‘pkg-config --libs opencv4‘ -pthread -std=c++17
```

for compiling. To run it there are different parameters to use:

1. `videoPath`: path of video to detect. There are some video for testing on directory `./media`

2. `k`: float parameter that lives between 0 and 1 to give the percentage of pixels that is used as threshold to detect if a frame is similar or not with respect the background. If $k = 0.2$ a frame with less than 20% of pixels different with respect the background one is considered similar to it.

3. `dataParalelizationType`: string that could be `SEQUENTIAL`, `PARALLEL` or `FASTFLOW` with respect the implementation that is decided to use to optimize the detector

4. `streamParalelizationType`: string that could be `SEQUENTIAL`, `PARALLEL` or `FASTFLOW` with respect the implementation that is decided to use to optimize the pipeline during the streaming of frames. If is used `PARALLEL` or `FASTFLOW` string the number of workers used for this parallelization is 3

5. `nw`: number of workers used to parallelize the data parallel tasks (grayscale, smoothing and detectDifference)

6. `formatter`: string that if is setted to `CSV` print log results in a csv formatted way (used for statistics). Otherwise if the string is not present or different log are more readable

To run the program the formatted command is:

```
./build/main [videoPath] [k] [dataParalelizationType]
     [streamParalelizationType] [nw] [formatter]
```

An example of this could be:

```
./build/main ./media/test_1m.mp4 0.4 SEQUENTIAL FASTFLOW 4
```