**SPM@Unipi-2016/2017**

**Project Report [Video Filtering]**

By: Ahmad Alleboudy

**Code repository:**

<https://github.com/alleboudy/spm>

**What has been implemented in the project**

Applying either Sobel filter or a contrast stretching filter on a given video, implemented in different fashions:

1. A pipeline using fastflow’s ordered farm [parallel2.cpp]
2. A pipeline using fastflow’s Parallel For [parallel3.cpp]
3. Same filtering implemented in a sequential manner [sequential.cpp]
4. Same filtering implemented using threads [threaded.cpp]
5. Same filtering implemented using C++11 future and promise [future.cpp]
6. Others, but not reported due to obvious overhead [combining the parallel farm together with a parallel for inside the workers of the farm, allowing each worker of the farm to have numberOfCores-(NumberOfWorkers+2)/NumberOfWorkers cores in a parallel for that loops through the image pixels to carry out the worker’s job], this method always utilized the 24 cores and showed too much overhead, so, it was omitted.

Included as well a CMakeLists.txt file for building the source code along with the already built binaries under spm/build

Also, test.sh for running the binaries for different number of cores and reporting the time taken  
And, clean.sh for cleaning the build folder removing \*.avi files, cmake files and built binaries

**Possible major FastFlow pipeline design choices**

1. A fast flow Ordered Farm with a custom emitter and collector [would not have to worry about the frames ordering as it is already handled by the ordered farm]
2. A fast flow “Parallel for” that runs over the frame pixels
3. A combination [omitted due to obvious overhead]

**How to compile**

Will need to have OpenCV installed and the fastflow root folder present on the machine in the same directory where the project folder is residing. [please feel free to edit the CMakeLists.txt for better convenience]

Using CMake , navigate to the project directory spm

mkdir build

cd build

cmake ..

make

**To run the compiled Binaries**

**For the sequential**

./sequential path/to/input/video/file.mp4 path/to/output/video/file.mp4 sobel

**For the others**

./other path/to/input/video/file.mp4 path/to/output/video/file.mp4 NumberOfWorkers sobel

**Where**

**path/to/input/video/file.mp4** is where the input video resides on the desk

**path/to/output/video/file.avi** is where we wish to have the output video

**NumberOfWorkers** is the number of threads to use

**sobel** is for applying the sobel filter, other option is **stretch** for applying contrast stretching

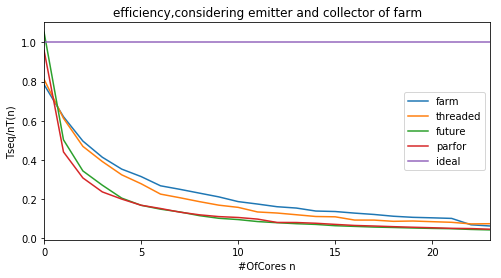
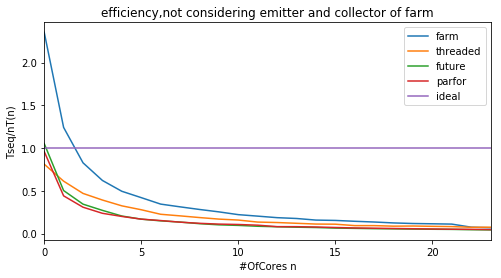
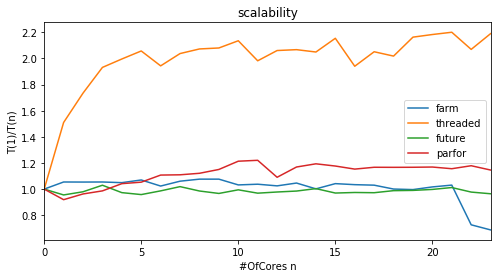
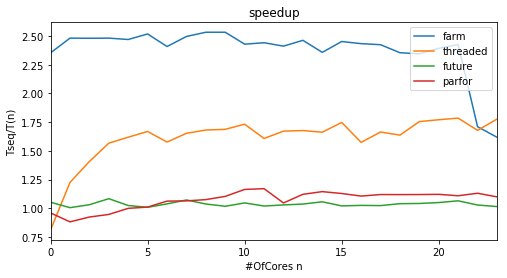
**A brief about the filters:**

**Sobel Filter:**Is used to enhance the edges in an image which benefits computer vision and image processing applications where edge detection is required  
<https://en.wikipedia.org/wiki/Sobel_operator>

**Contrast Stretching:**  
 Is used to limit the pixel intensities in an image between a given range [for simplicity here using min=0 and max =255, which forces the minimum intensity in the frame to be 0 and the max to be 255, meaning that if a frame has a min pixel intensity > 0, all of the pixels intensities will be pulled to the range 0-255]

<https://en.wikipedia.org/wiki/Normalization_(image_processing)>

**Performance Measures**

** **  
For the scalability it seems the farm implementation is not showing much benefit from using more cores to process the frames, my reasoning would be due to the bottleneck when saving the frames to disk, that is whatever the speed we can achieve from dividing the frames between workers, we are limited by the inevitable sequential parts of the pipeline(Amdahal Law ). [one frame write operation=~12ms, \*190 frames =2280ms which is around the time reported by the farm solution]****

We can notice a major drop in the farm implementation’s performance measures when the number of cores exceeds 22 [since two are already dedicated to the emitter and collector]