IMPROVING MACHINE LEARNING EXPERIENCE WITH MULTIMEDIA TECHNIQUES

Marco A. Franchi

ABSTRACT

It has been very common face machine learning algorithms at the multimedia area: traffic count; real-time vigilance cameras; baggage tracker; face/expression recognition; and so on. It is so common that it was designed a name for it, Machine Vision. However, it is perceptive the gap between machine learning and multimedia solutions, where even at simple embedded systems it is possible to reach 4k videos running at 60 frames per second, whereas the best neural network solution only handles 224x224 frames at 300 milliseconds in the same system. Due to this, a vast number of solutions as being developed: dedicated hardware for inference process; model manipulation; accelerated pre-processing image solutions; and video manipulation techniques. This paper is based at the study of these videos manipulation techniques, exposing the most common algorithmics, such as frame-skip, frame-droop, resizing, color convert, and overlay solutions; and showing the main difference between them, when the solution can be applied, and the expected performance for each one. This paper code can be find in GitHub, which includes a Jupyter Notebook reproducible version (https://github.com/marcofrk/ia369_final_project).

1. INTRODUCTION

Aiming to diminish the gap between the machine learning inference process and the multimedia capability, which reaches 4k@60fps, some video manipulation solutions was purposed. Among them, the most common is the overlay solutions, which are able to create alpha layers over the video and insert information on it. These overlays are very common on object detection algorithmics, once they are responsible for drawing a square, select or color an object at the scene. The most common overlays are Scalable Vector Graphics (SVG), Cairo, and OpenCV.

In addition to the overlays, it is important to use a great video framework able to handle all the elements involved at the inference and display solutions. One of the best and most useful ones is the GStreamer framework. GStreamer is able to handle plugins in pipelines, which is perfect to do quick tests. Apart from the video solutions, it is important to choose the best machine learning algorithmics as well.

With a focus on object detection, the most common and valued ones are Single Shot Detection (SSD) and Tensorflow. Both have an incredible inference process capability and the TFlite version demonstrated a great tool for embedded systems.

Thus, with all the tools chosen, this paper intends to compare the combination of theses algorithmics for object detection solutions. This comparison aims to demonstrate how we can increase the video frame rate with simple approaches and demonstrate the best scenarios to handle each neural network algorithmics and the overlays plugins behavior on these tests.

2. MATERIALS AND METHODS

This section describes the material such as the video files, models, labels, and programming language used, and the adopted methodology.

2.1. Programming Language

This paper uses Python 3 language and all the required support for the object detection algorithm: the overlays libs, GStreamer plugins, and TFLite.

OpenCV: Open Source Computer Vision Library, this library is cross-platform and free for user under open-source BSD license.

SVG image: Scalable Vector Graphics, a Extensible Markup Language (XML)-based vector image format for two-dimensional graphics.

GStreamer: GStreamer is a library for constructiong graphics of media-handling components and is released under the LGPL license.

TFLite: Tensoflow is a library for dataflow and differenctiable programming across of tasks, free and open-source under the Apache License 2.0. TFLite is a special version for mobile development.

SSD: Single Shot Detector algorithm designed for object detection in real-time applications.

2.2. Environment

At the paper repository, in addition to the source code, the repository provides a DockerFile for local running. This Docker image provides to run this same experiment with all the required packages and its validated versions. It is important to alert that this study was made to be performed on embedded Linux systems, including dedicated hardware for inference process and GPU comparison, but it was adapted to run on iterated notebooks, where even basics process such as video playback does not work properly.

2.3. Models and Labels

As the focus of this paper is the video techniques, and as the SSD and TFlite already have a huge numbers of pre-processed models, this paper will not care about pre-processing or training models and will uses pre-processed models available at the TFLite official object detection support web page.

For this, the following pre-tested model and labels will be used at the tests:

Model: mobilenet ssd v2 coco quant postprocess.tflite

Labels: coco labels.txt

2.3.1. MobileNets

MobileNets is a efficient models for mobile and embedded vision applications designed to effectively maximize accuracy while being mindful of the restricted resources for these devices.

2.3.2. Common Objects in Context(COCO) Labels

COCO is a large-scale object detection, segmentation, and captioning dataset.

2.4. GStreamer and V4L2

As mentioned before, this paper uses GStreamer framework to reproduce the videos files. For the comparison purpose, the following approaches will be performed:

* OpenCV V4L2 directly handle;
* GStreamer appsink pipeline + OpenCV V4L2 output;
* GStreamer appsink + appsrc pipelines;
* GStreamer SVG overlay plugins support.

The workflow section describes the difference between each process.

2.5. Workflow

The most basic Object Detection workflow can be described as an input Layer (an image, for instance), being processed by some Hidden Layers (model), and resulting in an Output Layer (square over an object and a label on it). So basically a model was used to identify some objects in an image.

However, this basic workflow will only work if the input layer has the same image size and color format expected for the model, i.e, in order to perform an Object Detection, an image pre-processing is required. Also, if it is expected to display the results in the same image, so an image pos-processing is required as well.

To become it more complex, take a video file as source instead of an image, and before considering each frame as an image, the video will require a decoding process, and than the frames will be pre-processed, calculated the results by the model, pos-processed, and than displayed with the label results.

Thus, we can split the Video Object Detection workflow into the following five steps:

1. Input data;
2. Image Pre-processing;
3. Inference proce
4. Image Pos-processing;
5. Output data.

2.5.1. OpenCV V4L2 directly handle

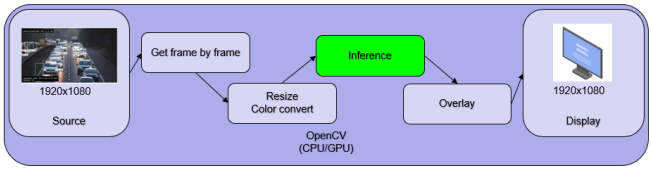

Figure 1: OpenCV V4L2 directly handle multimedia example.

The easiest way to perform Video Object Detection is by using OpenCV to perform the entire workflow process:

1. OpenCV uses Video4Linux (V4L2), a collection of device drivers and an API for supporting realtime video capture on Linux systems, which enable the user to pass a video file, image, or camera as Input Frames. OpenCV will handle it without any improvements.

2. OpenCV will take one-by-one Input Frame, pre-processing it, which includes color convert and image scaling, creating a Pre-processed Frame.

3. Here, the Pre-processed Frame is sent to the model, and then the inference process return the Resulted Frame.

4. The Resulted Frame is manipulated again by the OpenCV, drawing the squares, label, font colors, and etc. Then, the Resulted Frame is again scaled and applied a color convert, returning a Post-processed Frame.

5. In the end, OpenCV uses V4L2 to display the Post-processed Frame at the screen.

The coast of this approach is the impossibility to improve the video capture/display, plus the fact of OpenCV has to handle all the five steps, being overloaded all the time.

2.5.2. GStreamer appsink pipeline + OpenCV V4L2 output

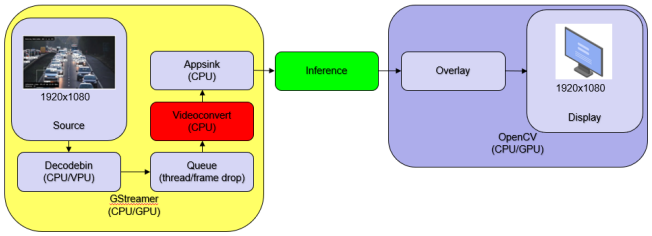

Figure 2: GStreamer appsink pipeline + OpenCV V4L2 output

As mentioned before, one of the issues from the OpenCV approach is the impossibility to improve the video capture. So this new approach will use a GStreamer pipeline in combination with OpenCV.VideoCapture function in order to improve the video input file. The GStreamer pipeline can be check below:

```
$ gst-launch-1.0 filesrc location=<video_file> ! qtdemux name=demux  demux.video_0 \
! queue ! decodebin ! queue max-size-buffers=1 ! videoconvert ! video/x-raw,format=BGR \
! appsink sync=false drop=True max-buffers=1 emit-signals=True max-lateness=8000000000
```

Figure 3: GStreamer appsink pipeline example

Although only the step 1 was changed - and then minus one step to be processed by OpenCV - this change allows us to use some interesting properties, such as dropping frame capability, the possibility to select the decoder plugin, and performing the videoconvert by GPU, when supported.

Note that the steps 2 can be avoided as well, by using the videoconvert and videoscale plugins. However, as it only supports CPU, the OpenCV shows a better performance than it.

As a result, the displayed framerate will not be impacted for the inference process time or the decoding process. However, the videoconvert usage, required for the appsink to be able to display the results at the screen, is a disadvantage, once it is only supported by CPU, and resize/color convert by CPU has a high processing coast.

2.5.3. *GStreamer appsink + appsrc pipelines*

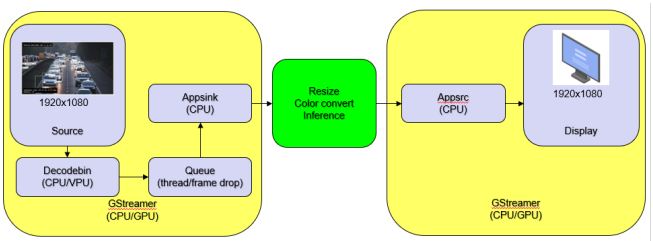

Figure 4: GStreamer appsink + appsrc pipelines

Following the idea of GStreamer usage instead of OpenCV, this approach shows the power of appsink and appsrc when used in combination.

By using the same GStreamer appsink pipeline from the last example, but this time sending the Post-processed Frame to another GStreamer pipeline instead of OpenCV, allows us to improve not only the video capture but the video displayed process as well:

```
$ gst-launch-1.0 filesrc location=<video_file> ! qtdemux name=demux  demux.video_0 \
! queue ! decodebin ! queue max-size-buffers=1 ! videoconvert ! video/x-raw,format=BGR \
! appsink sync=false drop=True max-buffers=1 emit-signals=True max-lateness=8000000000

$ gst-launch-1.0 appsrc name=src is-live=True block=True \
! video/x-raw,format=RGB,width=640,height=480, \
framerate=30/1,interlace-mode=(string)progressive \
! videoconvert ! ximagesink
```

Figure 5: GStreamer appsink and appsrc pipelines example

Thus, steps 1 and 5 are being processed by the GStreamer, and the OpenCV will handle only steps 2 and 4.

Again, note that steps 2 and 4 can be processed by GStreamer as well, but the OpenCV still shows a better performance than videoconvert and videoscale by CPU.

However, on embedded systems, with GPU plugins support to color convert and scaling, this combination shows very promisingly. The properties to be applied are almost unlimited and each one can perform a different result, and even the sink plugin can be one supported by GPU.

2.5.4. *GStreamer overlay plugins support*

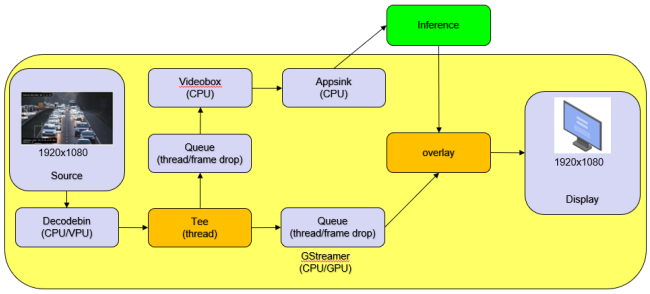

Figure 6: GStreamer overlay plugins support

As shown before, the image color convert and scaling can be an issue. Due to this, one additional property can be applied: overlay. As the name suggests, an overlay is the capability to display some planes over other planes, and GStreamer has support for a lot of overlay plugins.

With this approach, and by using rsvgoverlay plugin, this solution is able to take the Input Frame and send it to be processed and displayed at the same time. So when it was processed, the overlay, after includes all the image details, as square, labels, etc, will overlay the display image with all the contents:

```
$ filesrc location=<video_file> ! qtdemux name=demux  demux.video_0 \
! queue ! decodebin  ! videorate ! videoconvert n-threads=4 ! videoscale n-threads=4 \
! video/x-raw,width={width},height={height},framerate=30/1 ! queue max-size-buffers=1 leaky=downstream \
! tee name=t
    t. ! queue max-size-buffers=1 leaky=downstream ! videoconvert ! videoscale \
    ! video/x-raw,width={model_width},height={model_height ! videobox name=box autocrop=true
    ! video/x-raw,format=RGB,width={width},height={height} \
    ! appsink name=appsink emit-signals=true max-buffers=1 drop=true sync=fals

    t. ! queue ! videoconvert
    ! rsvgoverlay name=overlay ! videoconvert ! ximagesink
```

Figure 7: GStreamer SVG overlay pipeline example

Here, the tee usage creates two threads: one to be processed by the inference process; other to be displayed. The videobox keeps the frame and is able to return it to the overlay plugin, so what we see is the combination of two frames, one is the original video, without being touched, other is an alpha image with all the required resizing process being displayed over the original video.

With this approach, the video will never be impacted by the image pre-processing, inference, and image post-processing steps. So the framerate will be the highest possible.

And note again that all the plugins can be changed by the one with support to GPU, resulting in the best performance possible in terms of framerate.

3. RESULTS

The tests consist in execute one video on all the purposed solutions, running an Object Detection TFlite by using SSD algorithm. In order to test and compare the performance of each solution, two values are being evaluated: the video framerate (FPS) and the inference time.

The FPS presents the capability to display as many frames as possible in one unique second, while the inference time is the required elapsed time to get the frame, interpret it with TFLite, and return the objects founded to be displayed. The idea is to show that even for a slow inference time process, the user experience will not be affected and the displayed results will be smooth.

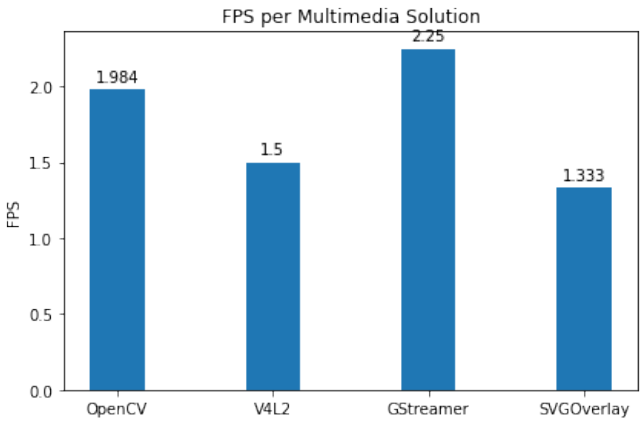

Figure 8: FPS per Multimedia Solution Results

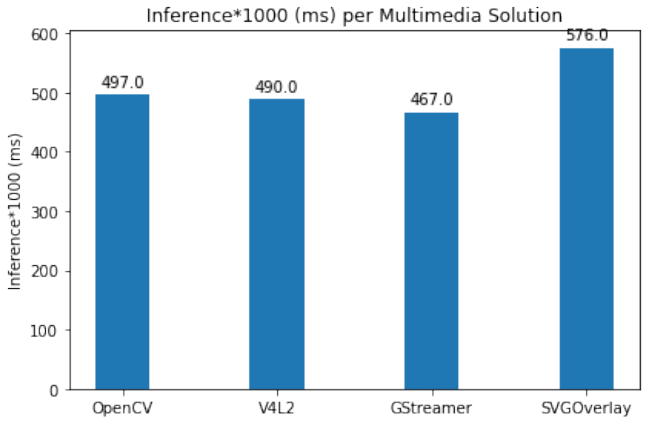

Figure 9: Inference*1000 (ms) per Multimedia Solution Results

4. CONCLUSION

Unfortunately, the notebooks have no performance enough even to handle playback videos, so one process as the object detection algorithms, which consumes even more process, was inviable to notebooks.

Plus, according to the values obtained, noticed that the number of about 0.5 milliseconds is far from the expected for a video object detection solution. With some simples calculations, it is easy to note that 0.5 ms of inference time represents 2 frames being calculated per second. It is very slow and it is far from the multimedia capabilities, which can achieve 60 frames per second.

However, one point was very clear: OpenCV V4L2 directly had a very poor performance than in comparison to the SVG Overlay performance, and this is exactly what this study intended to show, and all these solutions can be applied and enhanced with GPU supports.

5. REFERENCES

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A System for Large-Scale Machine Learning.

Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. 2017. Real-Time Video Analytics: The Killer App for Edge Computing.

Google. 2019. Coral Edge TPU. Retrieved June 16, 2020, from https://coral.ai/

Nvidia. 2020. NVIDIA DeepStream SDK. Retrieved June 16, 2020 from https://developer.nvidia.com/deepstream-sdk

GStreamer. 2019. GStreamer: open source multimedia framework. Retrieved June 16, 2020 from https://gstreamer.freedesktop.org

Intel. 2020.03. Intel's Deep Learning Inference Engine Developer Guide. Retrieved June 16, 2020 from https://docs.openvinotoolkit.org/latest/