video-pipeline

Oct 26, 2019

Contents

Simplify the video streaming pipeline to provide frame by frame image manipulation in near real-time.

Video streaming and image processing are really interesting!

This package aims to simplify the video streaming pipeline so users can focus on more interesting parts of image processing. To learn more about how this is accomplished and the details that make up the pipeline see *Architecture*. To begin using using this package right away, see *Getting Started* or *Examples*.

CHAPTER 1

Table of Contents

1.1 Getting Started

The `video-pipeline` comes with a command line interface (CLI) that you can utilize to preview, transport, and/or modify video streams!

1. First you need to install the `video-pipeline` module from PyPI by running:

```
pip install video-pipeline
```

2. Once installed `video-pipeline` should be on your `PATH`.
3. Make sure you have vlc installed and on your `PATH`.
4. Run the following command to start streaming video from your webcam:

```
video-pipeline start --source os --transport tcp-server transport-host=0.0.0.0␣
↪transport-port=8000
```

5. On the same computer (or another computer on your LAN) run the following command replacing `HOSTNAME` with the hostname of the computer running `video-pipeline`.

   Note: If you're running on a linux machine you can run `hostname` to get your `HOSTNAME`.

```
vlc "tcp/mjpeg://@HOSTNAME:8000/"
```

6. You should now see a stream of your webcam!

To learn more about the `video-pipeline` command line interface run `video-pipeline --help`. To see some more examples on how the CLI could be used see the *provided examples*.

If you have any issues questions, comments, or concerns please feel free to submit an issue to the issue tracker.

1.2 Architecture

video-pipeline consists of four high level components that describe how the flow of video data progresses through the system.

- VideoStream - Stream video frames from a video source and buffers the frames into memory.
- VideoProcessor - Processes image frames in parallel with a specified FrameFilter.
- FrameFilter - Transforms one image frame of video at a time. Used in the VideoProcessor.
- VideoTransport - Transport handler to stream content to a destination (i.e. disk/network).

These components and this document should give you a high level understanding of how video-pipeline works. Be aware that the core purpose of video-pipeline is to make it easier to understand how video processing works providing some deeper level of insight and not necessarily to be the fastest most performant solution. In some situations video-pipeline is even less performant than a custom solution. Depending on your requirements it might be worth the effort to implement lower level packages or even utilize C/C++ to reach higher performance benchmarks. To get a good overview of the performance scenarios of video-pipeline see the Performance section.

A lot of the inner workings and infrastructure in video-pipeline builds off the hard work of these fantastic lower level packages and their communities.

- imageio
- picamera
- scikit-image
- numpy

These components are separated so that each component could potentially run in parallel. "Potentially" because part of the scope of this project is to benchmark the various configurations that could be used. And also to help define clear boundaries between capturing, processing, and transporting video data.

The architecture itself follows a producer/consumer model where-as the VideoStream acts as the "Producer" and the rest of the components in the pipeline (ie VideoProcessor, VideoTransport) act as the "Consumer(s)". Meaning that when an image frame is produced from a VideoStream source the frame is passed along to the next component in the pipeline. This was done to follow a more intuitive streaming approach to keep the design as explicit as possible, while not making it too cumbersome to reason with.

When the VideoStream "produces" an image frame it's not really pushing that new frame to the next part of the pipeline but rather the image frames being captured are stored and then retrieved later by the pipeline itself. This allows the VideoStream to run semi-independently from the rest of the system and allows the pipeline to govern the speed in which it samples image frames. Letting the sample rate help stabilize the pipeline and mitigate backpressure issues.

After the pipeline samples image frames from the VideoStream the image frames are then passed to the VideoProcessor. The VideoProcessor can also accept a FrameFilter allowing the client to process/manipulate image frames in real-time. To learn more about the other types of FrameFilters see the Filters section. The VideoProcessor accepts each frame on the same execution context as the the pipeline and does not process them right away, instead the frames are queued up allowing for different modes of processing. These processing modes determine which execution context the image frame is processed in. The execution context is purposefully vague to indicate that it doesn't matter whether we're using Threads or Processes or the current execution context of the pipeline - it's up to the client!

Once the VideoProcessor is done processing an image frame, the processed frame is appended to an output queue called the "frame buffer". It is then the responsibility of the pipeline to pick up the processed image frames and pass them along to a VideoTransport. This transport component defines the basic structure for handling a processed image frame and follows the same model. This again allows the pipeline to drive the delivery of image frames based on the needs of the other components in the pipeline. The primary types of VideoTransports are TcpVideoTransport and FileVideoTransport. The TcpVideoTransport allows image frames to

be transferred to another machine over a TCP Socket making it fairly easy to stream video data directly to another host machine. The `FileVideoTransport` on the other hand takes image frames and writes them to disk. All `VideoTransport` types can be configured with various settings defined by their respective classes.

The general flow of video data can be described more succinctly with this diagram.

1.3 Filters

A filter, for `video-pipeline` intents and purposes, can be thought of as an independent module that knows how to filter/manipulate/transform an image frame. The transformation logic changes by implementation.

When an image frame is sent to a filter for processing, the input frame is in 3 channel RGB format and will expect filters to return a processed image with the same format.

1.3.1 Built-In Filters

The built-in filters are mostly built off of scikit-image and intentionally don't do much processing to keep them simple to provide a general idea of how one might go about building their own.

However some of the filters use opencv to demonstrate more complex filters. These filters will require you to install opencv since this package doesn't directly depend on it.

Available implementations:

- `gray-scale` [`GrayScaleFameFilter`] - Converts image frame to a gray scale image frame.
- `find-edges` [`FindEdgesFrameFilter`] - Filters out edges in an image frame.
- `color-filter` [`ColorFilterFrameFilter`] - Filters an image frame by a range of colors in HSL colorspace using opencv.
- `face-tracker` [`FaceTrackerFrameFilter`] - Draws bounding boxes around faces and eyes using the default opencv haarcascade frontalface classifier.

1.3.2 Custom Filters

Currently Not Implemented

To learn how to build your own custom filter see the *custom filter example*.

1.4 VideoProcessor

Manages the processing of `VideoStream`'s buffered frames through the use of `FrameFilters` to transform image frames in real-time.

Available implementations:

- `ParallelVideoProcessor` - follows a parallel processing model to process image frames in parallel to make sure they are processed as fast as possible.
- `SerialVideoProcessor` - doesn't use any parallel processing model to process image frames and is executed in the same execution context as the pipeline.

1.4.1 ParallelVideoProcessor

This parallel processing model is achieved by spawning a series of frame processors (denoted in the diagram below as *F1*, *F2*, and *F3*) which utilize the `FrameFilter` that is provided to the `VideoProcessor` to handle the filter logic of a single image frame.

After an image frame has processed the image frame processor iterates through the currently executing image frame processors to see if we can buffer any processed image frames that might be ahead of us. If a completed frame processor is found it's processed image frame is put into the frame buffer. We repeat this process until we run into a frame processor that hasn't been completed or we run out of frame processors to check. The frame buffer is then later pulled by the pipeline for transport.

1.4.2 SerialVideoProcessor

Nothing unique. Puts frames to process in a queue that is subsequently processed by the pipeline's execution context.

After an image frame has processed the pipeline takes the processed image frame immediately.

1.5 VideoStream

Connects to a video source and transforms the source into a buffer queue of frames. The video source changes by implementation.

Below you will find a diagram of a `VideoStream` implementation using a `VideoFrameCollector` to collect image frames. This is pretty much how every `VideoStream` operates.

Available implementations:

- `FileVideoStream` - *WIP*
- `OsVideoStream` - Streams a virtual device node created by the OS.
- `PiVideoStream` - Streams video from a RaspberryPi camera.
- `YouTubeVideoStream` - *WIP*

1.6 VideoTransport

The final destination of the pipeline. Transports processed image frames to some destination specified by the client.

Available implementations:

- `FileVideoTransport` - Transports image frames to a file in MJPEG format
- `TcpVideoTransport` - Transports image frames over a TCP socket.
- `VisVisVideoTransport` - Transports image frames to a visvis preview window.

1.7 Performance

`video-pipeline` is a package designed to take advantage of the Python `multiprocessing` library in order to speed up image manipulations and filters applied to a PiCamera video stream in real time.

The intent of this approach is to spread the work of image manipulation operations across many processes to "parallelize" the work that is done on each frame. This concept holds under the assumption that the image manipulation operations do not depend on the order in which the frame is captured, so multiple frames can be processed in parallel and then ordered chronologically later after completing the processing step.

Using `video-pipeline` to process many frames in parallel is supposedly capable of processing more image frames than processing each frame sequentially (capture->process->send out->repeat) although the code architecture needed to support `multiprocessing` is much more complex and likely introduces additional overhead to the pipeline versus serial processing. The true performance gains of using `video-pipeline` have not yet been quantified.

1.7.1 Testing approach

With this set of tests, I will quantify the performance of `video-pipeline` versus serial image processing on a PiCamera video stream. I will also quantify the operational overhead of the pipeline itself compared to no image processing. The primary metric for performance is frames per second (fps) of the video output.

Isolate image processing performance

I am interested only in the image processing part of `video-pipeline`'s performance, so I will create a "control" script that uses `video-pipeline` interfaces to PiCamera and outputting a video stream to a client. The control script will NOT use `video-pipeline` tools to handle images captured from PiCamera, but it WILL use the same operations on each frame, processing each frame directly and in order. In other words, the control script will be used to quantify the performance of the test setup itself and establish a baseline.

Quantify performance impacts from overhead

While the main benefit to using `video-pipeline` is its multiprocessing support, it is possible to run `video-pipeline` with one process. This effectively forces `video-pipeline` to operate on frames in series. While this is not a realistic use-case of the package, it provides us with an opportunity to quantify performance losses from any additional overhead introduced from using this package versus plain serial processing. Ideally, there would be little to no overhead and using a single-process `video-pipeline` would have the same impact as any other script between capturing and displaying frames.

Quantify gains from parallel processing

`video-pipeline` allows the user to select an arbitrary number of parallel processes to use for image processing. Clearly the upper limit to the number of concurrent parallel processes is limited by hardware capabilities, but we can still assess the performance gains compared to a single-process baseline. Even on hardware with one CPU core, the Python `multiprocessing` module abstracts this away so we can specify an arbitrary number of processes. For these tests we will compare the performance of `video-pipeline` with 1, 2, 4, 8, and 16 processes in the pool. The expectation is that performance improves with more than one process but with diminishing returns as the number of processes increases.

Try various image processing operations

The performance of an image processor is heavily dependent on the operations it must perform on each frame. As an image processing task has more operations or more complex computations to be run on every pixel, it is expected to have lower throughput (fps). As such, I will subject the serial baseline and `video-pipeline` to the following operations: - **No-op**. Output frames exactly match the captured frames. Any performance losses are attributed to overhead. - **Grayscale filter**. Convert captured frames (RGB) to single-channel grayscale, then output the grayscale frame as an equivalent 3-channel (RGB) frame. The number of operations is proportional to the number of pixels in the frame. This method is built in to PIL. - **Sobel filter**. Compute Sobel edge detection algorithms on the captured frame and output the filtered, grayscale result as an equivalent 3-channel (RGB) frame. The number of operations is proportional to 8x the number of pixels in the frame since it convolves a 3x3 kernel with every pixel. This method is built in to PIL. - **Color select filter**. Convert captured frames (RGB) to HWV color space. Create a binary mask of the pixels that are within the desired HSV bounds. Apply the binary mask to the original frame as a logical-and, then output the result as a 3-channel RGB frame. I don't know how many operations this is but it's probably more than the Sobel filter. These methods are built in to OpenCV.

1.7.2 Baseline Script

```
# TODO@phil: write me
# the source for the baseline serial, single process image processor will go here
```

1.7.3 Test Execution

TODO@phil explain the scene and how the test is conducted. use gifs where applicable.

1.7.4 Test Results

TODO@phil: include plots showing diminishing returns from increasing number of processes TODO@phil: include gif of sample video

Raspberry Pi 2

640x480	# processes	No-op	Grayscale	Sobel	Color Select
Baseline	1	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	1	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	2	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	4	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	8	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	16	?? fps	?? fps	?? fps	?? fps

Raspberry Pi 3 B+

640x480	# processes	No-op	Grayscale	Sobel	Color Select
Baseline	1	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	1	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	2	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	4	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	8	?? fps	?? fps	?? fps	?? fps
`video-pipeline`	16	?? fps	?? fps	?? fps	?? fps

1.8 RaspberryPi

Instructions to get `video-pipeline` setup on a RaspberryPi.

1.8.1 Dependencies

- `python3.7`
- `pip`

1. Install Python3.7 dependencies

```
sudo apt-get update
sudo apt-get install -y libffi-dev libbz2-dev liblzma-dev \
    libsqlite3-dev libncurses5-dev libgdbm-dev zlib1g-dev \
    libreadline-dev libssl-dev tk-dev build-essential \
    libncursesw5-dev libc6-dev openssl git

*Note*\ : If you don't install these you will run into issues regarding␣
↪pip
and/or the ssl module not being found.
```

2. Download and extract source

```
wget https://www.python.org/ftp/python/3.7.2/Python-3.7.2.tgz
tar zxvf Python-3.7.2.tgz
cd Python-3.7.2
./configure --enable-optimizations --disable-tests
```

3. Build and install Python 3.7

```
make -j 4 -l 4
sudo make install
```

4. Upgrade `pip`

```
sudo python3.7 -m pip install --upgrade pip
```

5. Install `picamera`.

```
sudo python3.7 -m pip install picamera
```

1.8.2 Installation

1. Install `video-pipeline`

```
sudo python3.7 -m pip install video-pipeline
```

2. Once installed you can use `video-pipeline` like you would on any other machine. See *Getting Started* for more information.

1.8.3 External Resources

- https://github.crookster.org/Installing-Python-3.7-Raspberry-Pi-Raspbian-stretch/
- https://unix.stackexchange.com/questions/190794/uninstall-python-installed-by-compiling-source

1.9 Examples

In this document you'll find some of the many ways you can utilize `video-pipeline`'s infrastructure. This document assumes you've gone through *Getting Started*.

1.9.1 Streaming with a Filter

Video streaming is relatively straight forward but sometimes I need to "filter" (aka pre-process) my image before streaming it.

The following example utilizes the built-in `gray-scale` filter to apply to every image frame in the hosted video stream.

1. Run the following command to start streaming video from your webcam through a `gray-scale` filter:

```
video-pipeline --host 0.0.0.0 --port 8000 --source os --filter gray-scale
```

2. Use `vlc` to view the video stream. Replacing `HOSTNAME` with your hostname:

```
vlc "tcp/mjpeg://@HOSTNAME:8000/"
```

1.9.2 Custom Filters

While it's nice to use the *built-in filters* of `video-pipeline` sometimes you need the ability to customize the filter's image manipulation logic.

The following creates a `always_coffee.py` filter that will be applied to every image frame in the hosted video stream.

1. Create a python script called `always_coffee.py` with the following:

   Note: The following uses scikit-image coffee .

```
from video_pipeline.frame_filter import FrameFilter
import skimage

class AlwaysCoffeeFrameFilter(FrameFilter):
    def process_frame(self, frame):
        return skimage.data.coffee()
```

2. In the same directory run the following command to start streaming video from your webcam through your custom filter by importing the script and specifying the filter:

```
video-pipeline start --source os --filter always_coffee.py --transport tcp-server␣
↪transport-host=0.0.0.0 transport-port=8000
```

3. Use `vlc` to view the video stream. Replacing `HOSTNAME` with your hostname:

```
vlc "tcp/mjpeg://@HOSTNAME:8000/"
```

1.10 video-pipeline

CHAPTER 2

Indices and tables

- genindex
- *Module Index*
- search