Flowty

Release 0.0.2

May 12, 2019

Contents

CHAPTER 1

Input/Output Formats

1.1 Input

Flowty uses the OpenCV VideoCapture API to load videos. It defaults to the FFmpeg backend.

Flowty is built to support reading from:

- H263
- H264
- VP9
- JPEG

For any of the video formats you can simply provide `/path/to/video.ext` as the `src` argument. To use a folder of sequentially numbered JPEGs as input you have to provide an image filename template such as `/path/to/video_dir/frame_%06d.jpg` (like specified in the `VideoCapture` constructor docs).

1.2 Output

Currently only one output format is supported: splitting flow into u, v pairs, quantising them and storing them as JPEGs. The `dest` argument must be a python format template with two interpolations: `{axis}` and `{index}`.

- `{axis}` will be replaced with `u` or `v` depending upon the direction of the flow field being written.
- `{index}` will be replaced by the flow frame index being written, typically you'll want to add some format specifiers to this to 0 pad it, e.g. `{index:06d}`.

An example template `dest` string is `/data/flow/{axis}/frame_{index:06d}.jpg`.

CHAPTER 2

Building Flowty

Flowty depends on OpenCV and FFmpeg. To build flowty outside of docker you will need to build and install these things yourself. This is surprisingly challenging given the finicky nature of OpenCV builds. It is also necessary to build OpenCV with CUDA support.

Your best bet for manually setting up an environment to run flowty is to look at the Dockerfile we build upon: willprice/opencv4. This is built on Ubuntu 18.04 (although very few changes are needed to go back to 16.04). You can see the flags we enable for building OpenCV. The key flags are:

- `OPENCV_GENERATE_PKGCONFIG=on` as we use `pkgconfig` in `setup.py` to get the OpenCV paths.
- `WITH_FFMPEG=on` since FFmpeg is the default backend
- `WITH_CUDA=on` as some of the algorithms are CUDA accelerated
- `OPENCV_EXTRA_MODULES_PATH=<path/to/opencv_contrib/modules>` since the `optflow` module isn't in core OpenCV.

Pay close attention to the output of `cmake` as the configuration step won't crash if FFmpeg isn't found, this will result in an OpenCV build incapable of reading videos (upon attempting to read a video it will just return no frames rather than raising an exception).

2.1 Resources

Check out the following

- The FFmpeg compilation guide
- The OpenCV compilation docs
- CUDA accelerated FFmpeg build

CHAPTER 3

Development

Development is best done running flowty inside of docker as this enables testing in its target environment. It also mitigates the need of setting up all the dependencies.

3.1 Using docker

The easiest way to hack on flowty is by mounting the flowty repository over the installed version in `/src` in the application container. e.g.

```
$ docker run -it --entrypoint bash \
    --runtime=nvidia \
    --mount type=bind,source=$PWD,target=/src \
    willprice/flowty
```

One has to be careful though as flowty is installed into the global environment, and the CLI application to `/usr/local/bin/flowty`, so once in the bash shell, run the following:

```
$ python3 -m pip uninstall --yes flowty
$ python3 -m pip install -e .[test]
$ pytest
```

The second `pip` command will install `flowty` in editable mode, that is, it will link directly to `/src` so when you make changes to any files, when you invoke the tests, they will run against the updated files.

Flowty (pronounced floti) is the swiss army knife of computing optical flow.

CHAPTER 4

Flow methods

The following OpenCV optical flow methods are implemented:

- TV-L1 (OpenCV reference CPU / GPU)

4.1 Roadmap

The following methods aren't implemented, but are on the roadmap to implement next.

- Brox (OpenCV reference GPU)
- Pyramidal Lucas-Kanade (OpenCV reference GPU)
- Farneback (OpenCV reference CPU / GPU)

CHAPTER 5

Usage

Flowty is packaged as a docker container to save you the hassle of having to build an accelerated version of OpenCV linked with FFmpeg by hand.

5.1 Dependencies

You will need the following software installed on your host:

- docker-ce
- nvidia-docker

To check these prerequisites are satified, run

```
$ docker run --runtime=nvidia --rm nvidia/cuda:10.1-base nvidia-smi
```

It should print the output of `nvidia-smi`

5.2 Invoking flowty

The container will run `flowty` by default, so you can provide arguments like you would if you were running `flowty` installed natively on your host.

For example, to compute TV-L1 optical flow from the video */absolute/path/to/video_dir/video.mp4* and save the flow u, v components as separate JPEGs in */absolute/path/to/video_dir/flow/{axis}/*, run the following command:

```
# CPU only version
$ docker run --rm -it \
    --mount "type=bind,source=/absolute/path/to/video_dir,target=/data" \
    willprice/flowty \
    tvl1 "/data/video.mp4" "/data/flow/{axis}/frame_{index:05d}.jpg"
```

(continues on next page)

(continued from previous page)

```
# GPU accelerated version
$ docker run --runtime=nvidia --rm -it \
    --mount "type=bind,source=/absolute/path/to/video_dir,target=/data" \
    willprice/flowty \
    tvl1 "/data/video.mp4" "/data/flow/{axis}/frame_{index:05d}.jpg" --cuda
```

Check out *Input/Output Formats* to find out more about supported formats.

5.3 Explanation

As docker isn't heavily used in the computer vision community we'll break the above example command down piece by piece to explain what's going on.

First, an explanation of what Docker is: Docker is a container platform, it allows you to build and run containers, which are a little like light-weight VMs. A container contains an entire OS and all the dependencies of the application you'd like to run. It doesn't share any storage with the host by default, so if you want to access files on your computer from inside the container you need to *mount* the directory into the container, this allows you to read and write to a host directory from within the container.

- `docker run` is used to run an *instance* of a container, this is the equivalent of launching a VM, but is much quicker.
- `--rm` indicates that we want to remove the container after usage (to prevent it taking up space)
- `-it` is short for `--interactive` and `--tty`. `--interactive` hooks up STDIN from your console to the container, allowing you to Ctrl-C to kill the operation, `--tty` allocates a pseudo-TTY and pipes this to STDOUT so that you can see the log messages printed by the tool, if this wasn't present no output from the container would be printed.
- `--runtime=nvidia` is used to switch out the docker container backend to NVIDIA's version which injects the necessary hooks to run CUDA programs. It is only necessary to specify this if you are using the CUDA accelerated routines (i.e. you pass the `--cuda` arg to flowty)
- `--mount type=bind,source=/absolute/path/to/video_dir,dest=/data` is the specification to docker that allows you to access a host directory within the container, this is called a *bind* mount, hence the `type=bind` specification. We mount the directory `/absolute/path/to/video_dir` on the host to `/data` within the container. Now anytime the container writes or reads anything from `/data` it will actually read from `/absolute/path/to/video_dir/`.