Flowty

Release 0.0.2

May 13, 2020

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.

We provide detailed instructions on using conda to build an environment suitable for flowty. We rely on CUDA 9.2+ due to constraints of the relatively modern compiler conda-forge use. If you need to use an older CUDA version, then you're best off compiling everything from scratch (including ffmpeg).

Contents

2.1 Building with conda

Warning: Note that we use `gcc` 7.3.0, which is fairly recent and only compatible with CUDA 9.2+. If you need to use a previous version, you're on your own and will have to build everything from scratch (the conda packages expect a recent C++ ABI and you won't be able to compile OpenCV and link to FFmpeg).

The first thing we need to do to produce a CUDA-equipped build of OpenCV is to find out what graphics card you have, and what driver you're running. This will define what PTX and BIN files we'll generate when building OpenCV.

```
$ nvidia-smi
Fri May 10 21:33:15 2019
```

(continues on next page)

(continued from previous page)

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 418.56       Driver Version: 418.56       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  Quadro K620         Off  | 00000000:01:00.0 Off |                  N/A |
| 34%   39C    P8     1W /  30W |      1MiB /  2001MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                       GPU Memory |
|  GPU       PID   Type   Process name                             Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

Now look up the max CUDA version support available for your driver: https://docs.nvidia.com/deploy/cuda-compatibility/index.html For 418.56, we can use up to CUDA 10.1 (as specified in the top right corner of `nvidia-smi`'s output, previous versions of `nvidia-smi` don't display this though).

We also see the graphics card is a Quadro K620, we need to look up its CUDA compute capability here: https://developer.nvidia.com/cuda-gpus. It lists 5.0 compute capability for this model, so we'll generate CUDA OpenCV binaries for this compute capability.

Now we need to set up our environment to build OpenCV and flowty. We'll assume you already have the following installed, if not install them before proceeding:

- `nvcc` (CUDA 9.2+, versions earlier than 9.2 don't support GCC 7.3.0, the compiler we'll use installed from conda-forge)
- `conda`

Hint: Any of the following variable definitions with `<something>` are placeholders and should be replaced based on your setup.

```
$ conda create -n flowty -c conda-forge python=3.6 pip ffmpeg lapack openblas \
    cmake libjpeg-turbo libpng zlib cython numpy pytest gxx_linux-64=7.3.0
$ conda activate flowty
$ export OPENCV_VERSION=4.1.0
$ export PKG_CONFIG_PATH=$CONDA_PREFIX/lib/pkgconfig:$CONDA_PREFIX/lib64/pkgconfig
$ export CC=x86_64-conda_cos6-linux-gnu-gcc CXX=x86_64-conda_cos6-linux-gnu-gcc
$ export CUDA_PATH="<path-to-cuda>"  # typically is /usr/local/cuda or similar
$ wget https://github.com/opencv/opencv/archive/${OPENCV_VERSION}.tar.gz \
    -O opencv-${OPENCV_VERSION}.tar.gz
$ wget https://github.com/opencv/opencv_contrib/archive/${OPENCV_VERSION}.tar.gz \
    -O opencv_contrib-${OPENCV_VERSION}.tar.gz
$ mkdir -p opencv/build
$ cd opencv
$ tar -xvf ../opencv-${OPENCV_VERSION}.tar.gz
$ tar -xvf ../opencv_contrib-${OPENCV_VERSION}.tar.gz
$ cd build
$ unset CXXFLAGS CFLAGS  # conda sets these by default when gxx_linux-64 is installed.
↪ These break .cu file compilation
$ export CUDA_COMPUTE_CAPABILITY=<your-gpu-compute-capability>  # 5.0 in our example.
$ cmake \
```

(continues on next page)

(continued from previous page)

```
    -D CMAKE_PREFIX_PATH="$CONDA_PREFIX" \
    -D CMAKE_BUILD_TYPE=Release \
    -D CMAKE_INSTALL_PREFIX=$CONDA_PREFIX \
    -D OPENCV_GENERATE_PKGCONFIG=ON \
    -D OPENCV_EXTRA_MODULES_PATH=../opencv_contrib-${OPENCV_VERSION}/modules \
    -D BUILD_opencv_optflow=ON \
    -D BUILD_opencv_cudaoptflow=ON \
    -D BUILD_opencv_tracking=ON \
    -D BUILD_opencv_calib3d=ON \
    -D BUILD_opencv_features2d=ON \
    -D BUILD_opencv_cudafeatures2d=ON \
    -D BUILD_opencv_flann=ON \
    -D BUILD_opencv_plot=ON \
    -D BUILD_opencv_highgui=ON \
    -D WITH_PNG=ON \
    -D WITH_FFMPEG=ON \
    -D WITH_CUDA=ON \
    -D WITH_CUBLAS=ON \
    -D WITH_OPENMP=ON \
    -D WITH_OPENCL=ON \
    -D WITH_LAPACK=ON \
    -D WITH_JPEG=ON \
    -D WITH_IPP=ON \
    -D WITH_MKL=ON \
    -D CUDA_TOOLKIT_ROOT_DIR="$CUDA_PATH" \
    -D CUDA_FAST_MATH=ON \
    -D CUDA_ARCH_PTX="$CUDA_COMPUTE_CAPABILITY" \
    -D CUDA_ARCH_BIN="$CUDA_COMPUTE_CAPABILITY" \
    -D ENABLE_FAST_MATH=ON \
    -D WITH_ADE=OFF \
    -D WITH_ARAVIS=OFF \
    -D WITH_CLP=OFF \
    -D WITH_EIGEN=OFF \
    -D WITH_GDAL=OFF \
    -D WITH_GDCM=OFF \
    -D WITH_GPHOTO2=OFF \
    -D WITH_GTK=OFF \
    -D WITH_ITT=OFF \
    -D WITH_JASPER=OFF \
    -D WITH_LIBREALSENSE=OFF \
    -D WITH_MFX=OFF \
    -D WITH_OPENEXR=OFF \
    -D WITH_OPENGL=OFF \
    -D WITH_OPENNI=OFF \
    -D WITH_OPENNI1=OFF \
    -D WITH_OPENVX=OFF \
    -D WITH_PROTOBUF=OFF \
    -D WITH_PTHREADS_PF=OFF \
    -D WITH_PVAPI=OFF \
    -D WITH_QT=OFF \
    -D WITH_QUIRC=OFF \
    -D WITH_TBB=OFF \
    -D WITH_TIFF=OFF \
    -D WITH_V3L=OFF \
    -D WITH_VA=OFF \
    -D WITH_VA_INTEL=OFF \
    -D WITH_VTK=OFF \
```

(continues on next page)

(continued from previous page)

```
    -D WITH_VULKAN=OFF \
    -D WITH_WEBP=OFF \
    -D WITH_XIMEA=OFF \
    -D WITH_XINE=OFF \
    -D WITH_HALIDE=OFF \
    -D WITH_GSTREAMER=OFF \
    -D WITH_V4L=OFF \
    -D BUILD_EXAMPLES=OFF \
    -D BUILD_DOCS=OFF \
    -D BUILD_TESTS=OFF \
    -D BUILD_PERF_TESTS=OFF \
    -D BUILD_opencv_apps=OFF \
    -D BUILD_opencv_aruco=OFF \
    -D BUILD_opencv_bgsegm=OFF \
    -D BUILD_opencv_bioinspired=OFF \
    -D BUILD_opencv_cudabgsegm=OFF \
    -D BUILD_opencv_cudaobjdetect=OFF \
    -D BUILD_opencv_cudastereo=OFF \
    -D BUILD_opencv_datasets=OFF \
    -D BUILD_opencv_dnn=OFF \
    -D BUILD_opencv_dnn_objdetect=OFF \
    -D BUILD_opencv_dpm=OFF \
    -D BUILD_opencv_face=OFF \
    -D BUILD_opencv_fuzzy=OFF \
    -D BUILD_opencv_gapi=OFF \
    -D BUILD_opencv_hfs=OFF \
    -D BUILD_opencv_img_hash=OFF \
    -D BUILD_opencv_java_bindings_generator=OFF \
    -D BUILD_opencv_js=OFF \
    -D BUILD_opencv_legacy=OFF \
    -D BUILD_opencv_line_descriptor=OFF \
    -D BUILD_opencv_ml=OFF \
    -D BUILD_opencv_objdetect=OFF \
    -D BUILD_opencv_phase_unwrapping=OFF \
    -D BUILD_opencv_photo=OFF \
    -D BUILD_opencv_python3=OFF \
    -D BUILD_opencv_python_bindings_generator=OFF \
    -D BUILD_opencv_quality=OFF \
    -D BUILD_opencv_reg=OFF \
    -D BUILD_opencv_rgbd=OFF \
    -D BUILD_opencv_saliency=OFF \
    -D BUILD_opencv_shape=OFF \
    -D BUILD_opencv_stereo=OFF \
    -D BUILD_opencv_stitching=OFF \
    -D BUILD_opencv_stitching=OFF \
    -D BUILD_opencv_structured_light=OFF \
    -D BUILD_opencv_superres=OFF \
    -D BUILD_opencv_surface_matching=OFF \
    -D BUILD_opencv_text=OFF \
    -D BUILD_opencv_videostab=OFF \
    -D BUILD_opencv_xfeatures2d=OFF \
    -D BUILD_opencv_xobjdetect=OFF \
    -D BUILD_opencv_xphoto=OFF \
    ../opencv-${OPENCV_VERSION}
```

Note: Note that most of the `cmake` build options in the above console session are disabling additional features

of OpenCV unused by flowty; if these cause errors, then feel free to drop the `OFF` options, they're just to speed up compilation time and save space.

Once configured, you need to double check the `cmake` output for FFmpeg, checking it was found. If you get something like this...

```
--   Video I/O:
--     DC1394:                      NO
--     FFMPEG:                      NO
--       avcodec:                   NO
--       avformat:                  NO
--       avutil:                    NO
--       swscale:                   NO
--       avresample:                NO
```

... then `cmake` has been unable to resolve the location of FFmpeg headers and libs. You should be aiming for something like this:

```
--   Video I/O:
--     DC1394:                      NO
--     FFMPEG:                      YES
--       avcodec:                   YES (58.35.100)
--       avformat:                  YES (58.20.100)
--       avutil:                    YES (56.22.100)
--       swscale:                   YES (5.3.100)
--       avresample:                YES (4.0.0)
```

Typically this is as a result of the `ffmpeg` pkgconfig files not being installed, or present within a directory on the `$PKG_CONFIG_PATH`.

Finally `make` and `make install`:

```
$ make -j $(nproc)
$ make install
```

Now you should have all the dependencies installed ready to build flowty.

```
$ mkdir flowty && cd flowty
$ export FLOWTY_VERSION=0.0.2
$ wget https://github.com/willprice/flowty/archive/v${FLOWTY_VERSION}.tar.gz \
    -O flowty.tar.gz
$ tar -xvf flowty.tar.gz
$ cd flowty-${FLOWTY_VERSION}
$ python setup.py build_ext --inplace
```

You will probably need to add the `$CONDA_PREFIX/lib64` directory to your `$LD_LIBRARY_PATH` as this is where opencv will have installed its shared libraries. Without setting this you will get error like...

```
ImportError: libopencv_core.so.4 .1: cannot open shared object file: No such file or␣
↪directory
```

Resolve this like so:

```
$ export LD_LIBRARY_PATH=$CONDA_PREFIX/lib:$CONDA_PREFIX/lib64
```

Check you can run the tests without failures:

```
$ PYTHONPATH=src pytest tests
```

If you were able to run the tests without any import failures, congratulations, you're now ready to install flowty and compute some flow!

```
$ python setup.py install
$ flowty --help
```

2.2 Building on Ubuntu

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).

Once you've managed to build OpenCV and install it, then have a look at the flowty dockerfile to see how to build and install flowty. Basically it's just:

```
$ pip3 install Cython numpy pytest
$ python setup.py build_ext --inplace
$ python setup.py install
```

2.3 Resources

For more, check out...

- 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)
- Brox (OpenCV reference GPU)
- Pyramidal Lucas-Kanade (OpenCV reference GPU)
- Farneback (OpenCV reference CPU / GPU)
- Dense Inverse Search (OpenCV reference CPU)
- Variational Refinement (OpenCV reference CPU)

4.1 Roadmap

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

- PCA flow (OpenCV reference CPU)
- Robust local optical flow (OpenCV reference CPU)

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/`.