KWIVER Documentation

Release 1

Kitware, Inc.

Sep 14, 2017

Contents

Contents:

CHAPTER 1

Introduction

The Kitware Image and Video Exploitation and Retrieval (KWIVER) toolkit is a collection of software tools designed to tackle difficult image and video analysis problems and other related challenges. KWIVER is an ongoing effort to transition technology developed over multiple years by Kitware's computer vision group to the open source domain in order to further research, collaboration, and product development.

KWIVER contains the following components.

'VITAL'_ A core library of abstractions and data types used by various KWIVER components. Major elements of VITAL are: - Basic data types used throughout Kwiver. - Provides abstract algorithm interfaces for implementations in the ARROWS component. - Configuration support library providing a common approach to run time configuration of the components. - An OS abstraction layer that provides system services in a platform independent manner. - flexible logging support that can interface to different logging back ends. - General purpose plugin architecture.

'Stream Processing Toolkit (sprokit)'_ Sprokit is the "Stream Processing Toolkit", a library aiming to make processing a stream of data with various algorithms easy. It supports divergent and convergent data flows with synchronization between them, connection type checking, all with full, first-class Python bindings.

Sprokit also contains a set of processes and example pipelines that support basic operations such as image and video input and display, wrappers for common algorithms.

'ARROWS'_ ARROWS is an open source C++ collection of algorithms for making measurements from aerial video. Initial capability focuses on estimating the camera flight trajectory and a sparse 3D point cloud of the scene.

Additionally, a separate repository, Fletch, is a CMake based project that assists with acquiring and building common Open Source libraries useful for developing video exploitation tools.

There is no single "correct" way to build KWIVER. Rather, depending on your use case you will configure and build KWIVER in ways that make the tools and libraries you require avaialable to you. In this documentation we'll detail and document some of the more common and useful usecases.

Video Analytics Toolchain

Coming Soon!

SMQTK/C++ Bridge

The SMQTK C++ Bridge is a mechanism that allows C++ based programs to make calls to the Python based SMQTK project. In particular, it allows a C++ program to call the SMQTK descriptor engine by providing an image as input and receiving a feature vector in return.

The bridge is based on Kitware's KWIVER project. In particular it uses the data types and core services provided by KWIVER's VITAL project and the pipeline processing (including Python compute nodes) provided by SMQTK's Sprokit project. The *Block Diagramm of the SMQTK/C++ Bridge* shows the structure of the solution.

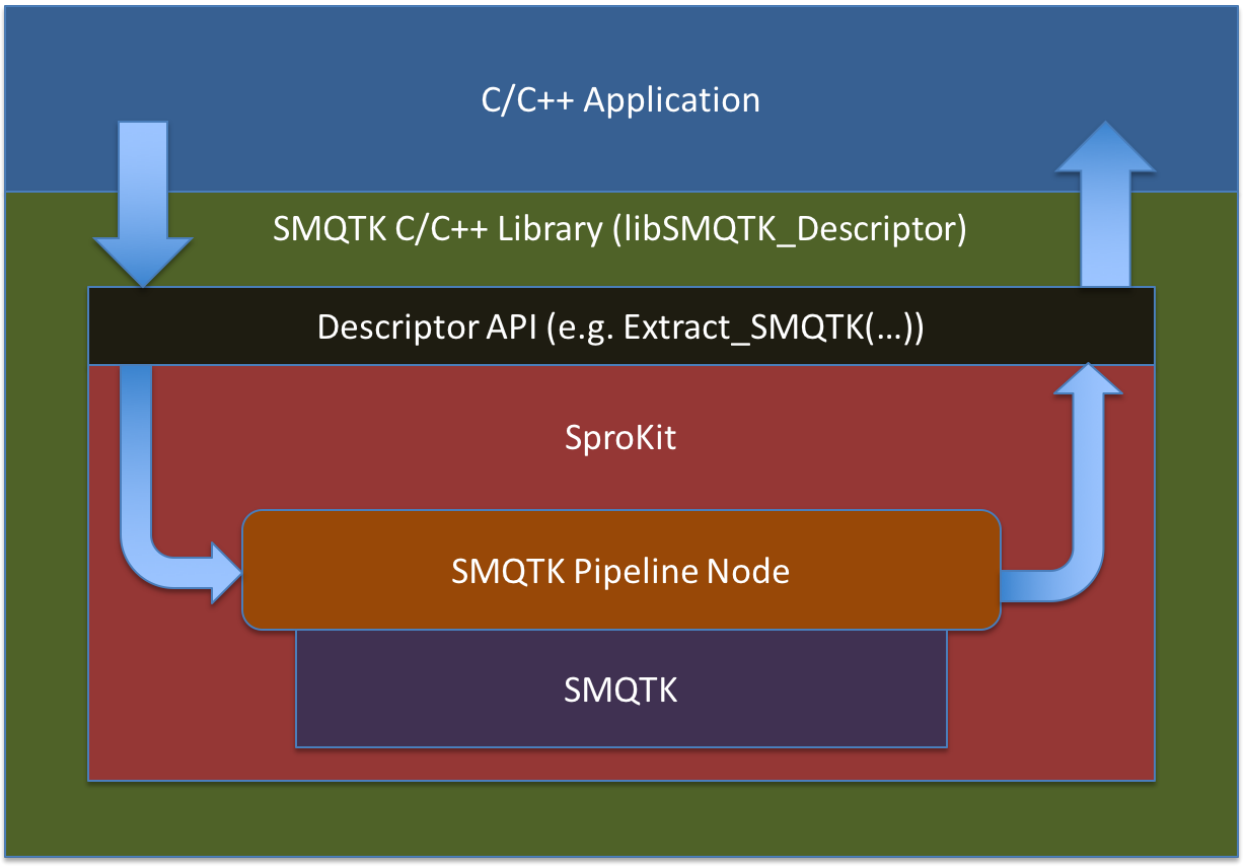

Fig. 1.1: *Block Diagramm of the SMQTK/C++ Bridge*

The KWIVER repository and its associated build framework takes care of building VITAL and Sprokit once it's properly configured. Also, in order to build KWIVER, you must build some of the third party dependencies maintained by Kitware's Fletch project (a repository that is used to manage the build of a variety of computer vision and machine learning tools).

This document will help you configure Fletch and KWIVER so that you can use them with SMQTK and Python. It will also discuss the C++/SMQTK bridge and how you can integrate it into your projects.

The instructions assume that you will set up a directory structure similar to the *SMQTK/C++ Bridge Source Organization* shown.

These projects are interdependent, so you'll want to fetch the repositories and checkout the correct branch for all of them first.

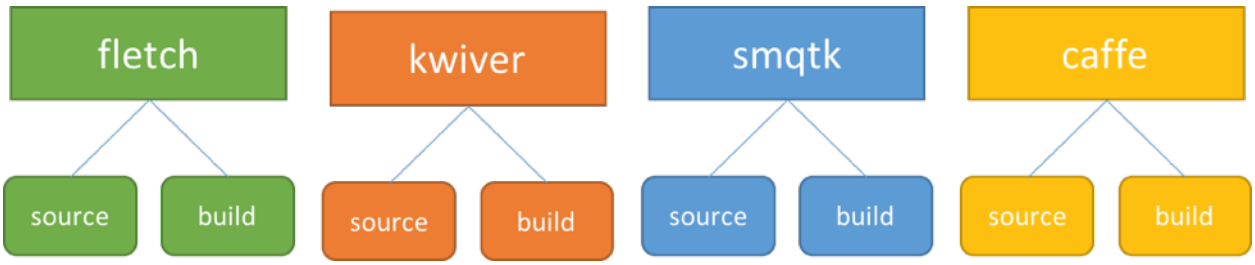

Fig. 1.2: *SMQTK/C++ Bridge Source Organization*

Getting the Code

Fletch

In the Fletch directory:

```
git clone https://github.com/Kitware/fletch.git source
cd source
git checkout master
```

KWIVER

In the KWIVER directory:

```
git clone https://github.com/kitware/kwiver.git source
cd source
git checkout master
```

SMQTK

In the SMQTK directory:

```
git clone https://github.com/Kitware/SMQTK.git source
cd source
git checkout v0.2
```

CAFFE

In the CAFFE directory:

```
git clone https://github.com/BVLC/caffe.git source
cd source
git checkout rc2
./scripts/download_model_binary.py models/bvlc_reference_caffenet
./data/ilsvrc12/get_ilsvrc_aux.sh
```

For CAFFE, in addition to obtaining the source code, we're fetching some pre-trained models that we can use.

Setting up a Python Environment

In order to use the SMQTK/C++ bridge, you must have a Python environment on your system. We recommend installing the Miniconda environment from Continuum – an open source Python environment that makes it very easy to set up Python for scientific computing.

Make sure that the Miniconda python command is the first one in your PATH:

```
export PATH=~/.miniconda/bin:${PATH}
```

We will also need to create an SMQTK conda environment in which we will run SMQTK:

```
conda create -n smqtk --file smqtk/source/requirements.conda.txt
source activate smqtk
pip install -r smqtk/source/requirements.pip.txt
pip install scikit-image
pip install protobuf
```

Building the Code

Fletch

Note: It is important that only the Miniconda environment (and not the smqtk environment is active when building fletch. Run `source deactivate` to be sure.)

From your Fletch directory:

```
mkdir build
cd build
cmake -C ../source/sprokit/processes/examples/call_SMQTK_pipeline/fletch-precache.
↪cmake ../source/
```

This will configure Fletch to build the projects that KWIVER needs to build properly for use with SMQTK/C++ bridge.

To actually build Fletch execute the command:

```
cmake --build .
```

(Note that there is period (**.**) at the end of that command)

KWIVER

Note: It is important that the SMQTK Miniconda environment is active when building fletch. Run `source activate smqtk` to be sure.

What follows are the steps required to build KIWVER to provide the SMQTK/C++ bridge

In the KWIVER directory:

```
source activate smqtk
mkdir build
cd build
```

To configure the build:

```
cmake -Dfletch_DIR:PATH=../../fletch/build/ -C ../source/sprokit/processes/examples/
↪call_SMQTK_pipeline/kwiver-precache.cmake ../source/
```

Verify that the PYTHON specifications are correct (assuming you installed miniconda in ~/miniconda):

```
PYTHON_EXECUTABLE            ~/miniconda/bin/python
PYTHON_INCLUDE_DIR           ~/miniconda/include/python2.7
PYTHON_LIBRARY               ~/miniconda/lib/libpython2.7.so
```

And finally, build KWIVER:

```
make
make install
```

CAFFE

One of the feature's of SMQTK is that it can use a CAFFE based CNNN as a descriptor. In general, you simply need to build CAFFE with it's Python bindings turned on:

```
mkdir build
cd build
cmake -DBOOST_ROOT:PATH=../../fletch/build/install/  -DOpenCV_DIR:PATH=../../fletch/
↪build/install/share/OpenCV/ ../source/
make
make install
```

SMQTK

From the SMQTK directory:

```
mkdir build
cd build
cmake ../source
```

Testing the Code

Set your PATH to include the new projects. The following commands will set the environment so the examples can be run:

```
source kwiver/build/install/setup_KWIVER.sh
source smqtk/build/setup_env.build.sh
export PYTHONPATH=${PWD}/caffe/build/install/python:${PYTHONPATH}
```

To test that the the SMQTK/C++ bridge is working, we will run the SMQTK_Descriptor_test application. This application is an example C++ application that accepts a configuration file to specify the location of your CAFFE installation model, files and GPU configuration and a single image or list of images and submit that image (or images) to SMQTK to have CAFFE compute the descriptor. The application does nothing with the descriptor other than print

the first 50 elements. This is a sample program only. You may wish to read the source code for the program in `kwiver/source/examples/call_SMQTK_pipeline` to see how you can adapt this technique to your own programs.

To use the application, you'll need to provide it with a configuration file that tells it what descriptor to use.

Here is an example configuration file that uses SMQTK's CaffeDesriptorGenerator to run the ImageNet CNN and use the 7th layer as a descriptor:

```
{
        "CaffeDescriptorGenerator": {
        "batch_size": 256,
        "data_layer": "data",
        "gpu_device_id": 0,
        "image_mean_filepath": "/path/to/caffe/source/data/ilsvrc12/imagenet_mean.
↪binaryproto",
        "load_truncated_images": true,
        "network_is_bgr": true,
        "network_model_filepath": "/path/to/caffe/source/models/bvlc_reference_
↪caffenet/bvlc_reference_caffenet.caffemodel",
        "network_prototxt_filepath": "/path/to/caffe/source/models/bvlc_reference_
↪caffenet/deploy.prototxt",
        "return_layer": "fc7",
        "use_gpu": false
        },
"type": "CaffeDescriptorGenerator"
}
```

The command to run the test application is as follows (all one line, run from the kwiver/build directory):

> LD_PRELOAD=~/miniconda/envs/smqtk/lib/libpython2.7.so.1.0 ./examples/call_SMQTK_pipeline/SMQTK_Descriptor_test my-caffe-config ../source/examples/cat.jpg

---

Note: The "LD_PRELOAD" variable setting is required to make sure that sprokit does not try to use the system python instead of the SMQTK python environment you created. We will be working to make this unecessary as soon as possible.

When the application runs, it will eventually print out lines that look like this (after many log messages):

```
Descriptor size: 4096
0 0 0 0 0 0 3.53588 0 0 0 0 3.3475 0 0 0 0 1.67483 0 0 0 0 0 0 0 0 7.86536 0 0 0 0 0␣
↪0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Which represents the first 50 elements of the returned feature vector. If all of the elements are the value `0.223`, then something went wrong and the feature vector was not run

If you wish to use a different model, say AlexNet, make sure you run `./scripts/download_model_binary.py models/bvlc_alexnet` in the Caffe source directory and then you can create a different configuration file that specifies `/path/to/caffe/source/models/bvlc_alexnet/bvlc_alexnet.caffemodel` for `network_model_filepath` and `/path/to/caffe/source/models/bvlc_alexnet/deploy.prototxt` for `network_prototxt_filepath`.

Using the SMQTK/C++ Bridge in Your Code

The C++ interface to SMQTK descriptors is through the SMQTK_Descriptor class as defined in the include file `SMQTK_Descriptor.h`.

The SMQTK descriptor API is built as part of kwiver and is available in the library `libSMQTK_Descriptor.so`. You will need to add this to your build instructions as `-lSMQTK_Descriptor` or in an equivalent manner appropriate for your build system.

The class provides a single method to apply the descriptor to an image and return the descriptor vector, which is described as follows:

```
std::vector< double > ExtractSMQTK(  cv::Mat cv_img, std::string const& config_file );
```

cv_img An image in OpenCV format

config_file The name of the SMQTK descriptor configuration file in JSON format. See *Testing the Code* for details.

The ExtractSMQTK() method is synchronous in that it will return with the descriptor vector even though the descriptor calculation may be multi-threaded.

The source code for the `call_SMQTK_pipeline` provides an example of how to use this call in your own programs:

Include the file SMQTK_Descriptor.h in the source code to get the interface to the SMQTK_Descriptor class, as shown below:

```
#include "SMQTK_Descriptor.h"
```

The following two source statements implement and apply the descriptor:

```
kwiver::SMQTK_Descriptor des; // Create object
std::vector< double > results = des.ExtractSMQTK( img, file_name );
```

The inputs are the OpenCV format image and the name of the descriptor configuration file. The output is the descriptor vector of doubles.

A sample program is provided in the source file SMQTK_Descriptor_test.cxx which serves as a test of the API and an example of how it is used. The operation of this test program is discussed above.

CHAPTER 2

Installing Kwiver

These instructions are designed to help build Kwiver on a fresh machine. They were written for and tested on Ubuntu 16.04 Desktop version. Other Linux machines will have similar directions, but some steps (particularly the dependency install) may not be totally identical.

Install Dependencies

Some of the dependencies required for Kwiver can be installed with one quick and easy instruction with no configuration required. Different Linux distributions may have different packages already installed, or may use a different package manager than apt, but even on Ubuntu this should help to provide a starting point.

sudo apt-get install git zlib1g-dev libcurl4-openssl-dev libexpat1-dev dh-autoreconf liblapack-dev libxt-dev

sudo apt-get build-dep libboost-all-dev qt5-default

Install CMAKE

The version of cmake you currently get with apt is too old to use for kwiver, so you need to do a manual install. Go to the cmake website, *https://cmake.org/download*, and download the appropriate binary distribution (for Ubuntu, this would be something like cmake-3.6.1-Linux-x86_64.sh, depending on version). Download the source code, cmake-3.6.1.tar.gz (or just download and use the installer for windows). To untar and build the source, use the following set of commands. Keep in mind that if you're not using version 3.6.1, you'll need to update the version number to match your download.

cd ~/Downloads

tar zxfv cmake-3.6.1.tar.gz

cd cmake-3.6.1

./bootstrap –system-curl –no-system-libs

make

sudo make install

sudo ln -s /usr/local/bin/cmake /bin/cmake

These instructions build the source code into a working executable, installs the executable into a personal directory, and then lets the operating system know where that directory is so it can find cmake in the future.

Install Fletch

Fletch is a CMake driven build that will help configure and install a series of component packages necessary for Kwiver, like Eigen and Boost. Navigate to the directory where you want to put your source code and builds. I personally like to use ~/Work and then set up a new directory for each repo. With all dependencies for Fletch installed in the last couple of steps, Fletch should build without any issues.

mkdir fletch

cd fletch

git clone https://github.com/kitware/fletch.git

mkdir build

cd build

cmake -Dfletch_ENABLE_ALL_PACKAGES:bool=on ../fletch

make

Install Kwiver

After Fletch is built, you should have everything necessary to build Kwiver. Navigate back to the directory you want to put Kwiver in (if you followed the directions above, the command to return is *cd ../..*). In the cmake step, make sure to fill in your Fletch build directory so Kwiver knows where to find its dependencies. For example, I would use *cmake -Dfletch_DIR:path=/home/dave/Work/fletch/build ../kwiver*.

mkdir kwiver

cd kwiver

git clone https://github.com/kitware/kwiver.git

mkdir build

cd build

cmake -Dfletch_DIR:path=<fletch_build_directory> ../kwiver

make

CHAPTER 3

Indices and tables

- genindex
- modindex
- search