What it is?

- OpenCV 4.0 comes with an experimental Graph API module (see opencv/modules/gapi). This is a new API which allows to enable offload and optimizations for image processing / CV algorithms on pipeline level.
- The idea behind G-API is to declare image processing task in form of expressions and then submit it for execution – using a number of available backends. At the moment, there's reference "CPU" (OpenCV-based), "GPU" (also OpenCV T-API-based), and experimental "Fluid' backends available, with other backends coming up next.
- G-API is an uncommon OpenCV module since it acts as a framework: it provides means for declaring operations, building graphs of operations, and finally implementing the operations for a particular backend. G-API model enforces separation between interfaces and implementations, so once an algorithm is expressed in G-API terms, it can be scaled/ported/offloaded to a new platform easily.
- G-API CPU (OpenCV) backend implements G-API standard functions using OpenCV itself (core/imgproc modules) and acts as a quick prototyping/porting/testing backend. If you have an image processing algorithm composed of OpenCV-like functions already, you would be able to switch quickly to G-API by using this backend.
- G-API Fluid backend implements a cache-efficient execution model and allows to save memory dramatically – e.g. a 1.5GB image processing pipeline fits into 750KB memory footprint with G-API/Fluid. G-API comes with a number of operations implemented for Fluid backend, so one can switch OpenCV/Fluid operations within a graph easily and even mix both in the same graph.
- G-API GPU backend implements the majority of available functions and allows to run OpenCL kernels on available OpenCL-programmable devices. At the moment, GPU backend is based on OpenCV Transparent API; in future versions it will be extended to support integration of arbitrary OpenCL kernels (and likely be renamed to "OpenCL backend").
- G-API ONNX backend implements ONNX models inference operations on input data and outputs the results. At the moment, ONNX backend is based on ONNX Runtime C/C++ API.

Building G-API

G-API is built with OpenCV by default, however some features may require additional options or dependencies enabled.

Building with OpenVINO Toolkit support

- Get `openvino` from https://docs.openvinotoolkit.org/
- Configure enviroment:

```
source path-to-unpacked-openvino/bin/setupvars.sh
```

- Build G-API with OpenVINO support:

```
cmake /path-to-opencv -DWITH_INF_ENGINE=ON
```

NOTE: Set `INF_ENGINE_RELEASE` to the proper version, depending on the package you use.

Building with PlaidML support

- Follow instruction to build PlaidML2: https://plaidml.github.io/plaidml/docs/building

```
bazelisk build //plaidml2:wheel
pip install -U bazel-bin/plaidml2/wheel.pkg/tmp/dist/*
```

- Setup PLAIDML_DEVICE

```
python3 /path-to-plaidml/plaidml/plaidml2/plaidml_setup.py
export `cat ~/.plaidml2`
```

- Build G-API with PlaidML2 support:

```
cmake /path-to-opencv -DPlaidML2_DIR=path-to-miniconda3/share/plaidml2 -DWITH_PLAIDML=ON
```

- Run tests:

```
/path-to-opencv-build/bin/opencv_test_gapi --gtest_filter=*GAPI_PlaidML_Pipelines*
```

Building with Microsoft ONNX Runtime support

- Build and install the ONNX RT (currently tested with v1.5.1):

```
$ git clone --recursive https://github.com/microsoft/onnxruntime.git
$ cd onnxruntime
$ git checkout v1.5.1
$ git submodule update --init
$ ./build.sh --config Release --build_shared_lib --parallel \
$     --cmake_extra_defines CMAKE_INSTALL_PREFIX=install
$ cd build/Linux/Release
$ make install
```

- Then specify extra options to OpenCV CMake:

```
$ cmake /path-to-opencv -DWITH_ONNX=ON -DORT_INSTALL_DIR=/path-to-ort-install-dir
```

Building with oneVPL Toolkit support

- Build (https://github.com/oneapi-src/oneVPL) or install `oneVPL` from installation manager (https://www.intel.com/content/www/us/en/developer/tools/oneapi/onevpl.html)
- Source an oneVPL environment variables or execute shell script

```
<installation path>/share/oneVPL/env/vars.bat"
```

- Then specify extra options to OpenCV CMake:

```
$ cmake /path-to-opencv -DWITH_GAPI_ONEVPL=ON
```

- Run tests:

```
/path-to-opencv-build/bin/opencv_test_gapi --gtest_filter=*OneVPL_Source*
```

- How to run example and configure `oneVPL` file-based source and launch `OpenVINO` inference please find out examples in `/path-to-opencv-build/bin/example_gapi_onevpl_infer_single_roi`. Also see `Building with OpenVINO Toolkit support` section to how to configure G-API with `OpenVINO`

VPL Source capabilities & limitations

- G-API oneVPL Source implements string-based parameters configuration mechanism through `CfgParam` objects packed into array or initialization list. These parameters has reflection of `oneVPL` configuration parameters which can be found by the link https://spec.oneapi.io/versions/latest/elements/oneVPL/source/programming_guide/VPL_prg_session.html#dsp-conf-prop-table. Some of these parameters are MAJOR and some others are OPTIONAL. Using MAJOR params is necessary to make VPL source work, while OPTIONAL params provide extra optimization tricks or advice VPL dispatcher to select the preferable oneVPL library implementation (like a version index) and so on. All params have `name` and `value` fields which should be mapped to VPL-related configuration parameter by G-API oneVPL Source by itself.

Lets consider example of choosing Hardware Acceleration type for VPL Source:

As described in https://spec.oneapi.io/versions/latest/elements/oneVPL/source/programming_guide/VPL_prg_session.html#dsp-conf-prop-table it has name `mfxImplDescription.AccelerationMode` and with type `MFX_VARIANT_TYPE_U32` then we just use

```
std::vector<CfgParam> cfg_params;
cfg_params.push_back(CfgParam::create_acceleration_mode(MFX_ACCEL_MODE_VIA_D3D11));
```

or

```
std::vector<CfgParam> cfg_params;
cfg_params.push_back(CfgParam::create_acceleration_mode("MFX_ACCEL_MODE_VIA_D3D11"));
```

G-API oneVPL Source interface must parse either `int` or `string` like parameter value representation. To find out which VPL parameters are supported please proceed by https://github.com/opencv/opencv/blob/4.x/modules/gapi/include/opencv2/gapi/streaming/onevpl/cfg_params.hpp#L63 (List of parameters is updated regularly)

- Only Windows platform is tested and supported with Hardware Acceleration DX11

According to `oneVPL` dispatcher the user is free to choose preferred acceleration type. But default deployment of VPL implementation driver supply hardware acceleration only which means that ALL decoding operations use hardware acceleration. It is possible only to clarify to G-API oneVPL Source what types of memory should produce oneVPL Source in its own `cv::MediaFrame` as result. If no parameters described `mfxImplDescription.AccelerationMode` have passed during G-API oneVPL source construction then `cv::MediaFrame` will carry CPU memory as frame data (it means copy from CPU to acceleration during decode operation and back again). In otherwise, let's assume we passed `CfgParam::create_acceleration_mode("MFX_ACCEL_MODE_VIA_D3D11"))`, a `cv::MediaFrame` would carry GPU memory as data in DX11Texture2D inside and would require using `cv::MediaFrame::access` to get it's value.

- G-API oneVPL Source support video decoding either using RAW video stream formats (see onVPL support codes) and inner demultiplexing using `Microsoft Foundation Primitives`.

Implementation doesn't rely on file extension to choose format because usually it might be wrong. Instead the following interface is provided: If no parameters described `mfxImplDescription.mfxDecoderDescription.decoder.CodecID` have passed during G-API oneVPL source construction then implementation try out demultiplexing schema; if specific codecId is set (for example `CfgParam::create_decoder_id(MFX_CODEC_HEVC)`) then implementation assume RAW stream unconditionally.

Please use environment variable `OPENCV_LOG_LEVEL=Info` at least to consider full description in case of any source file errors but usually default level `OPENCV_LOG_LEVEL=Warn` is enough

How-to-launch OpenVINO Inference using VPL Source practical guide

First of all make sure that conditions are met: https://github.com/opencv/opencv/wiki/Graph-API#building-with-openvino-toolkit-support and https://github.com/opencv/opencv/wiki/Graph-API#building-with-onevpl-toolkit-support. The sample with oneVPL & OpenVINO Inference Engine can be found in `gapi/samples` directory under openCV project directories tree. Only infer single ROI is supported at now.

(Current configuration parameters are obsolete and new will be introduced in https://github.com/opencv/opencv/pull/21716. Description written down in assumption that this PR was merged)

- Source video file is a RAW encoded video stream (h265 for example)

```
example_gapi_onevpl_infer_single_roi --facem=<model path> --
cfg_params="mfxImplDescription.mfxDecoderDescription.decoder.CodecID:MFX_CODEC_HEVC;" --
input=<full RAW file path>
```

Please explore the full list of supported codec constants here:
https://github.com/opencv/opencv/blob/4.x/modules/gapi/include/opencv2/gapi/streaming/onevpl/cfg_params.hpp#L95

- Source file is a containerized media file: *.mkv, *.mp4, etc. (Applicable for WINDOWS only)

```
example_gapi_onevpl_infer_single_roi --facem=<model path> --cfg_params="" --input=<full
RAW file path>
```

or

```
example_gapi_onevpl_infer_single_roi --facem=<model path> --input=<full file path>
```

Please pay attention that examples launch non-optimized pipeline for default acceleration types:

- VPL Source uses GPU device for decoding with copying media frame into CPU RAM
- VPL preprocessing used GPU device for decoding with copying media frame from/into CPU RAM
- Inference uses CPU device too

Also it is possible to configure such pipeline stages in fine-grained way and seize heterogenous computation advantages. Thus, three acceleration parameters exposed: `source_device` , `preproc_device` and `faced` . Variety combinations of either `CPU` & `GPU` values are supported. Full list of supported configuration enumerated in sample support device matrix and is constantly growing.

The most interesting cases are:

- Default GPU-accelerated decoding & copy-based CPU use-case is similar with (synonym for empty parameters):

```
example_gapi_onevpl_infer_single_roi  <...> --source_device=CPU --preproc_device=CPU --facem=CPU
```

- GPU decode/preprocessing pipeline with CPU-based inference:

```
example_gapi_onevpl_infer_single_roi  <...> --source_device=GPU --preproc_device=GPU --facem=CPU
```

- Full copy-free GPU pipeline can be configured as:

```
example_gapi_onevpl_infer_single_roi  <...> --source_device=GPU --preproc_device=GPU --facem=GPU
```

Testing G-API

By default, the OpenCV G-API comes with its own test suite (`opencv_test_gapi`). Note that extra (external) G-API modules may introduce their own test suites. G-API tests are built and run in a regular way:

Linux

```
$ make -j4 opencv_test_gapi
$ bin/opencv_test_gapi
```

Windows

```
$ cmake --build . --target opencv_test_gapi --config Release -- /maxcpucount:4
$ bin\Release\opencv_test_gapi.exe
```

Requirements

Some G-API tests require test data to be available. This data is taken from the opencv_extra repo. You have to set the `OPENCV_TEST_DATA_PATH` environment variable to avoid failed tests (due to absence of test data):

```
export OPENCV_TEST_DATA_PATH=/Linux/path/to/opencv_extra/testdata
```

or

```
SET OPENCV_TEST_DATA_PATH=\Windows\path\to\opencv_extra\testdata
```

Some tests require **external** test data to be available. This is test data not included in opencv_extra. ONNX models are an example. The absence of this data doesn't break the tests. Tests are skipped without **external** test data.

With OpenVINO Inference Engine

When you build G-API with OpenVINO Inference Engine support (`-DInferenceEngine_DIR=...` `-DWITH_INF_ENGINE=ON`), some extra tests for inference are enabled and require `OPENCV_DNN_TEST_DATA_PATH` to be set and **models downloaded** using the command below!

```
$ git clone https://github.com/openvinotoolkit/open_model_zoo.git
$ cd open_model_zoo/tools/model_tools
$ ./downloader.py -o /path/to/opencv_extra/testdata/dnn/omz_intel_models/ \
    --cache_dir /path/to/opencv_extra/testdata/dnn/.omz_cache/ \
    --name age-gender-recognition-retail-0013
$ export OPENCV_DNN_TEST_DATA_PATH=/path/to/opencv_extra/testdata/dnn
```

With ONNX Runtime

When you build G-API with ONNX Runtime support, tests for inference are enabled and require `OPENCV_GAPI_ONNX_MODEL_PATH` to be set:

```
$ export OPENCV_TEST_DATA_PATH=/path-to-opencv-extra/testdata
$ export OPENCV_GAPI_ONNX_MODEL_PATH=/path-to-onnx-models/
```

and models are downloaded using the commands:

```
$ git clone --recursive https://github.com/onnx/models.git
$ cd models
$ git lfs pull --include=path-to-desired-onnx-model --exclude=""
```

Submitting G-API PRs

G-API supports so-called STANDALONE mode build which is not validated by default. Please put the below lines to the PR description to validate that this mode is not broken by the PR:

```
force_builders=Custom,Custom Win,Custom Mac
build_gapi_standalone:Linux x64=ade-0.1.1f
build_gapi_standalone:Win64=ade-0.1.1f
build_gapi_standalone:Mac=ade-0.1.1f
build_gapi_standalone:Linux x64 Debug=ade-0.1.1f

Xbuild_image:Custom=centos:7
Xbuildworker:Custom=linux-1
build_gapi_standalone:Custom=ade-0.1.1f

build_image:Custom=ubuntu-openvino-2021.4.1:20.04
build_image:Custom Win=openvino-2021.4.1
build_image:Custom Mac=openvino-2021.4.1

test_modules:Custom=gapi,python2,python3,java
test_modules:Custom Win=gapi,python2,python3,java
test_modules:Custom Mac=gapi,python2,python3,java

buildworker:Custom=linux-1
# disabled due high memory usage: test_opencl:Custom=ON
test_opencl:Custom=OFF
test_bigdata:Custom=1
test_filter:Custom=*
```

In order to enable `oneVPL` related builders please use `Custom Win` with `build_image:Custom Win=gapi-onevpl-2021.6.0`

Notes:

- ADE version may change, refer to the latest correct one (see `DownloadADE.cmake`).

G-API build depends on a number of external components which may not be built by default, e.g. PlaidML:

```
force_builders=Custom
buildworker:Custom=linux-1
build_image:Custom=plaidml2
test_modules:Custom=gapi
test_filter:Custom=*ML*
```

or ONNX Runtime:

```
force_builders=Custom
buildworker:Custom=linux-1
build_image:Custom=ubuntu-onnx:20.04
test_modules:Custom=gapi
test_filter:Custom=*ONNX*
```

or FreeType:

```
force_builders=Custom
buildworker:Custom=linux-1
build_image:Custom=ubuntu-gapi-freetype:16.04
```

It is also worth testing if any internal API changes are made.

Materials

- A tutorial and some documentation chapters are already available!
- Check out these slides as a compact introduction to G-API.