OpenCV 4.4 Graph API

Overview and programming by example

Dmitry Matveev
Intel Corporation

September 4, 2020

Outline

G-API: What is, why, what's for?

OpenCV evolution in one slide

Version 1.x – Library inception

- Just a set of CV functions + helpers around (visualization, IO);

Version 2.x – Library rewrite

- OpenCV meets C++, `cv::Mat` replaces `IplImage*`;

Version 3.0 – Welcome Transparent API (T-API)

- `cv::UMat` is introduced as a *transparent* addition to `cv::Mat`;
- With `cv::UMat`, an OpenCL kernel can be enqeueud instead of immediately running C code;
- `cv::UMat` data is kept on a *device* until explicitly queried.

OpenCV evolution in one slide (cont'd)

Version 4.0 – Welcome Graph API (G-API)

- A new separate module (not a full library rewrite);
- A framework (or even a *meta*-framework);
- Usage model:
  - *Express* an image/vision processing graph and then *execute* it;
  - Fine-tune execution without changes in the graph;
- Similar to Halide – separates logic from platform details.
- More than Halide:
  - Kernels can be written in unconstrained platform-native code;
  - Halide can serve as a backend (one of many).

OpenCV evolution in one slide (cont'd)

Version 4.2 – New horizons

- Introduced in-graph inference via OpenVINO™ Toolkit;
- Introduced video-oriented Streaming execution mode;
- Extended focus from individual image processing to the full application pipeline optimization.

Version 4.4 – More on video

- Introduced a notion of stateful kernels;
  - The road to object tracking, background subtraction, etc. in the graph;
- Added more video-oriented operations (feature detection, Optical flow).

Why G-API?

Why introduce a new execution model?

- Ultimately it is all about optimizations;
  - or at least about a *possibility* to optimize;
- A CV algorithm is usually not a single function call, but a composition of functions;
- Different models operate at different levels of knowledge on the algorithm (problem) we run.

Why G-API? (cont'd)

Why introduce a new execution model?

- Traditional – every function can be optimized (e.g. vectorized) and parallelized, the rest is up to programmer to care about.
- Queue-based – kernels are enqueued dynamically with no guarantee where the end is or what is called next;
- Graph-based – nearly all information is there, some compiler magic can be done!

What is G-API for?

Bring the value of graph model with OpenCV where it makes sense:

- Memory consumption can be reduced dramatically;
- Memory access can be optimized to maximize cache reuse;
- Parallelism can be applied automatically where it is hard to do it manually;
  - It also becomes more efficient when working with graphs;
- Heterogeneity gets extra benefits like:
  - Avoiding unnecessary data transfers;
  - Shadowing transfer costs with parallel host co-execution;
  - Improving system throughput with frame-level pipelining.

Programming with G-API

G-API Basics

G-API Concepts

- Graphs are built by applying *operations* to *data objects*;
  - API itself has no "graphs", it is expression-based instead;
- Data objects do not hold actual data, only capture *dependencies*;
- Operations consume and produce data objects.
- A graph is defined by specifying its *boundaries* with data objects:
  - What data objects are *inputs* to the graph?
  - What are its *outputs*?

The code is worth a thousand words

```
#include <opencv2/gapi.hpp>                        // G-API framework header
#include <opencv2/gapi/imgproc.hpp>                // cv::gapi::blur()
#include <opencv2/highgui.hpp>                     // cv::imread/imwrite

int main(int argc, char *argv[]) {
    if (argc < 3) return 1;

    cv::GMat in;                                   // Express the graph:
    cv::GMat out = cv::gapi::blur(in, cv::Size(3,3)); // ‘out‘ is a result of ‘blur‘ of ‘in‘

    cv::Mat in_mat = cv::imread(argv[1]);          // Get the real data
    cv::Mat out_mat;                               // Output buffer (may be empty)

    cv::GComputation(cv::GIn(in), cv::GOut(out))   // Declare a graph from ‘in‘ to ‘out‘
        .apply(cv::gin(in_mat), cv::gout(out_mat)); // ...and run it immediately

    cv::imwrite(argv[2], out_mat);                 // Save the result
    return 0;
}
```

The code is worth a thousand words

Traditional OpenCV

```
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>

#include <opencv2/highgui.hpp>

int main(int argc, char *argv[]) {
    using namespace cv;
    if (argc != 3) return 1;

    Mat in_mat = imread(argv[1]);
    Mat gx, gy;

    Sobel(in_mat, gx, CV_32F, 1, 0);
    Sobel(in_mat, gy, CV_32F, 0, 1);

    Mat mag, out_mat;
    sqrt(gx.mul(gx) + gy.mul(gy), mag);
    mag.convertTo(out_mat, CV_8U);

    imwrite(argv[2], out_mat);
    return 0;
}
```

OpenCV G-API

```
#include <opencv2/gapi.hpp>
#include <opencv2/gapi/core.hpp>
#include <opencv2/gapi/imgproc.hpp>
#include <opencv2/highgui.hpp>

int main(int argc, char *argv[]) {
    using namespace cv;
    if (argc != 3) return 1;

    GMat in;
    GMat gx  = gapi::Sobel(in, CV_32F, 1, 0);
    GMat gy  = gapi::Sobel(in, CV_32F, 0, 1);
    GMat mag = gapi::sqrt(  gapi::mul(gx, gx)
                          + gapi::mul(gy, gy));
    GMat out = gapi::convertTo(mag, CV_8U);
    GComputation sobel(GIn(in), GOut(out));

    Mat in_mat = imread(argv[1]), out_mat;
    sobel.apply(in_mat, out_mat);
    imwrite(argv[2], out_mat);
    return 0;
}
```

The code is worth a thousand words (cont'd)

What we have just learned?

- G-API functions mimic their traditional OpenCV ancestors;
- No real data is required to construct a graph;
- Graph construction and graph execution are separate steps.

What else?

- Graph is first *expressed* and then *captured* in an object;
- Graph constructor defines *protocol*; user can pass vectors of inputs/outputs like

  `cv::GComputation(cv::GIn(...), cv::GOut(...))`
- Calls to `.apply()` must conform to graph's protocol

On data objects

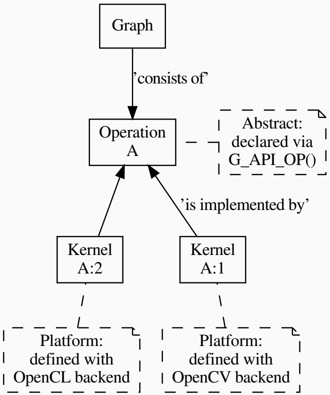
Graph protocol defines what arguments a computation was defined on (both inputs and outputs), and what are the shapes (or types) of those arguments:

Shape	Argument	Size
`GMat`	`Mat`	Static; defined during graph compilation
`GScalar`	`Scalar`	4 x `double`
`GArray<T>`	`std::vector<T>`	Dynamic; defined in runtime
`GOpaque<T>`	`T`	Static, `sizeof(T)`

`GScalar` may be value-initialized at construction time to allow expressions like `GMat a = 2*(b + 1)`.

On operations and kernels

- Graphs are built with Operations over virtual Data;
- Operations define interfaces (literally);
- Kernels are implementations to Operations (like in OOP);
- An Operation is platform-agnostic, a kernel is not;
- Kernels are implemented for Backends, the latter provide APIs to write kernels;
- Users can *add* their own operations and kernels, and also *redefine* "standard" kernels their own way.



On operations and kernels (cont'd)

Defining an operation

- A type name (every operation is a C++ type);
- Operation signature (similar to `std::function<>`);
- Operation identifier (a string);
- Metadata callback – describe what is the output value format(s), given the input and arguments.
- Use `OpType::on(...)` to use a new kernel `OpType` to construct graphs.

```
G_API_OP(GSqrt,<GMat(GMat)>,"org.opencv.core.math.sqrt") {
    static GMatDesc outMeta(GMatDesc in) { return in; }
};
```

On operations and kernels (cont'd)

`GSqrt` **vs.** `cv::gapi::sqrt()`

- How a type relates to a functions from the example?
- These functions are just wrappers over `::on`:

```
G_API_OP(GSqrt,<GMat(GMat)>,"org.opencv.core.math.sqrt") {
    static GMatDesc outMeta(GMatDesc in) { return in; }
};
GMat gapi::sqrt(const GMat& src) { return GSqrt::on(src); }
```

- Why – Doxygen, default parameters, 1:n mapping:

```
cv::GMat custom::unsharpMask(const cv::GMat &src,
                             const int       sigma,
                             const float     strength) {
    cv::GMat blurred   = cv::gapi::medianBlur(src, sigma);
    cv::GMat laplacian = cv::gapi::Laplacian(blurred, CV_8U);
    return (src - (laplacian * strength));
}
```

On operations and kernels (cont'd)

Implementing an operation

- Depends on the backend and its API;
- Common part for all backends: refer to operation being implemented using its *type*.

OpenCV backend

- OpenCV backend is the default one: OpenCV kernel is a wrapped OpenCV function:

```
GAPI_OCV_KERNEL(GCPUSqrt, cv::gapi::core::GSqrt) {
    static void run(const cv::Mat& in, cv::Mat &out) {
        cv::sqrt(in, out);
    }
};
```

Operations and Kernels (cont'd)

Fluid backend

- Fluid backend operates with row-by-row kernels and schedules its execution to optimize data locality:

```
GAPI_FLUID_KERNEL(GFluidSqrt, cv::gapi::core::GSqrt, false) {
    static const int Window = 1;
    static void run(const View &in, Buffer &out) {
        hal::sqrt32f(in .InLine <float>(0)
                     out.OutLine<float>(0),
                     out.length());
    }
};
```

- Note `run` changes signature but still is derived from the operation signature.

Operations and Kernels (cont'd)

Specifying which kernels to use

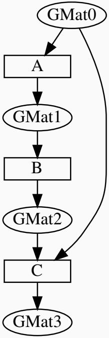
- Graph execution model is defined by kernels which are available/used;
- Kernels can be specified via the graph compilation arguments:

```
#include <opencv2/gapi/fluid/core.hpp>
#include <opencv2/gapi/fluid/imgproc.hpp>
...
auto pkg = cv::gapi::combine(cv::gapi::core::fluid::kernels(),
                             cv::gapi::imgproc::fluid::kernels());
sobel.apply(in_mat, out_mat, cv::compile_args(pkg));
```

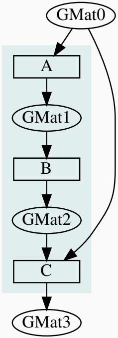
- Users can combine kernels of different backends and G-API will partition the execution among those automatically.

Heterogeneity in G-API

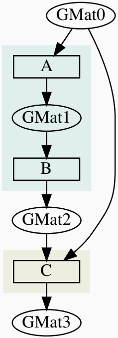
Automatic subgraph partitioning in G-API



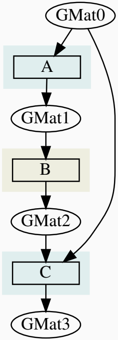
The initial graph: operations are not resolved yet.



All operations are handled by the same backend.



`A` & `B` are of backend `1`, `C` is of backend `2`.



`A` & `C` are of backend `1`, `B` is of backend `2`.

Heterogeneity in G-API

Heterogeneity summary

- G-API automatically partitions its graph in subgraphs (called "islands") based on the available kernels;
- Adjacent kernels taken from the same backend are "fused" into the same "island";
- G-API implements a two-level execution model:
  - Islands are executed at the top level by a G-API's Executor;
  - Island internals are run at the bottom level by its Backend;
- G-API fully delegates the low-level execution and memory management to backends.

Inference and Streaming

Inference with G-API

In-graph inference example

- Starting with OpencV 4.2 (2019), G-API allows to integrate `infer` operations into the graph:

```
G_API_NET(ObjDetect, <cv::GMat(cv::GMat)>, "pdf.example.od");

cv::GMat in;
cv::GMat blob = cv::gapi::infer<ObjDetect>(bgr);
cv::GOpaque<cv::Size> size = cv::gapi::streaming::size(bgr);
cv::GArray<cv::Rect>  objs = cv::gapi::streaming::parseSSD(blob, size);
cv::GComputation pipelne(cv::GIn(in), cv::GOut(objs));
```

- Starting with OpenCV 4.5 (2020), G-API will provide more streaming- and NN-oriented operations out of the box.

What is the difference?

- `ObjDetect` is not an operation, `cv::gapi::infer<T>` is;
- `cv::gapi::infer<T>` is a generic operation, where `T=ObjDetect` describes the calling convention:
  - How many inputs the network consumes,
  - How many outputs the network produces.
- Inference data types are `GMat` only:
  - Representing an image, then preprocessed automatically;
  - Representing a blob (n-dimensional `Mat`), then passed as-is.
- Inference backends only need to implement a single generic operation `infer`.

Inference with G-API

But how does it run?

- Since `infer` is an Operation, backends may provide Kernels implenting it;
- The only publicly available inference backend now is OpenVINO™:
  - Brings its `infer` kernel atop of the Inference Engine;
- NN model data is passed through G-API compile arguments (like kernels);
- Every NN backend provides its own structure to configure the network (like a kernel API).

Inference with G-API

Passing OpenVINO™ parameters to G-API

- `ObjDetect` example:

```
auto face_net = cv::gapi::ie::Params<ObjDetect> {
    face_xml_path,        // path to the topology IR
    face_bin_path,        // path to the topology weights
    face_device_string,   // OpenVINO plugin (device) string
};
auto networks = cv::gapi::networks(face_net);
pipeline.compile(.., cv::compile_args(..., networks));
```

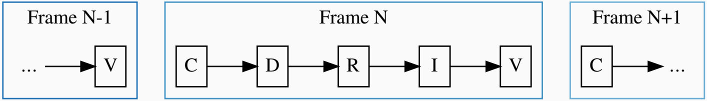
- `AgeGender` requires binding Op's outputs to NN layers:

```
auto age_net = cv::gapi::ie::Params<AgeGender> {
    ...
}.cfgOutputLayers({"age_conv3", "prob"}); // array<string,2> !
```

Streaming with G-API

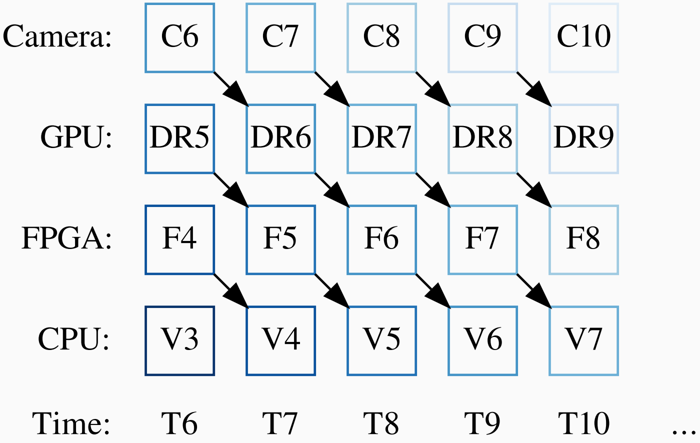
Anatomy of a regular video analytics application

Streaming with G-API



Serial execution of the sample video analytics application

Streaming with G-API



Pipelined execution for the video analytics application

Streaming with G-API: Example

Serial mode (4.0)

```
pipeline = cv::GComputation(...);

cv::VideoCapture cap(input);
cv::Mat in_frame;
std::vector<cv::Rect> out_faces;

while (cap.read(in_frame)) {
    pipeline.apply(cv::gin(in_frame),
                   cv::gout(out_faces),
                   cv::compile_args(kernels,
                                    networks));
    // Process results
    ...
}
```

Streaming mode (since 4.2)

```
pipeline = cv::GComputation(...);

auto in_src = cv::gapi::wip::make_src
    <cv::gapi::wip::GCaptureSource>(input)
auto cc = pipeline.compileStreaming
    (cv::compile_args(kernels, networks))
cc.setSource(cv::gin(in_src));
cc.start();

std::vector<cv::Rect> out_faces;
while (cc.pull(cv::gout(out_faces))) {
    // Process results
    ...
}
```

More information

https://opencv.org/hybrid-cv-dl-pipelines-with-opencv-4-4-g-api/

Latest features

Latest features

Python API

- Initial Python3 binding is available now in `master` (future 4.5);
- Only basic CV functionality is supported (`core` & `imgproc` namespaces, selecting backends);
- Adding more programmability, inference, and streaming is next.

Latest features

Python API

```
import numpy as np
import cv2 as cv

sz  = (1280, 720)
in1 = np.random.randint(0, 100, sz).astype(np.uint8)
in2 = np.random.randint(0, 100, sz).astype(np.uint8)

g_in1 = cv.GMat()
g_in2 = cv.GMat()
g_out = cv.gapi.add(g_in1, g_in2)
gr    = cv.GComputation(g_in1, g_in2, g_out)

pkg   = cv.gapi.core.fluid.kernels()
out   = gr.apply(in1, in2, args=cv.compile_args(pkg))
```

Understanding the "G-Effect"

Understanding the "G-Effect"

What is "G-Effect"?

- G-API is not only an API, but also an *implementation*;
  - i.e. it does some work already!
- We call "G-Effect" any measurable improvement which G-API demonstrates against traditional methods;
- So far the list is:
  - Memory consumption;
  - Performance;
  - Programmer efforts.

Note: in the following slides, all measurements are taken on Intel® Core™-i5 6600 CPU.

Understanding the "G-Effect"

Memory consumption: Sobel Edge Detector

- G-API/Fluid backend is designed to minimize footprint:

Input	OpenCV MiB	G-API/Fluid MiB	Factor Times
512 x 512	17.33	0.59	28.9x
640 x 480	20.29	0.62	32.8x
1280 x 720	60.73	0.72	83.9x
1920 x 1080	136.53	0.83	164.7x
3840 x 2160	545.88	1.22	447.4x

- The detector itself can be written manually in two `for` loops, but G-API covers cases more complex than that;
- OpenCV code requires changes to shrink footprint.

Understanding the "G-Effect"

Performance: Sobel Edge Detector

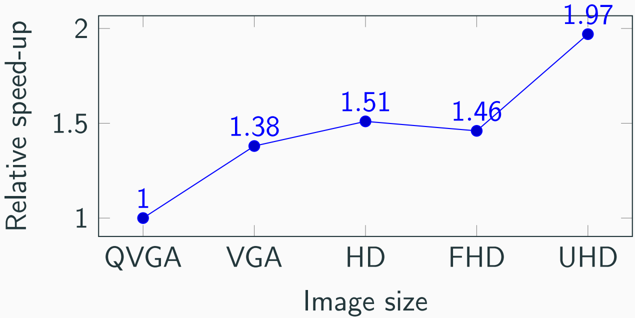
- G-API/Fluid backend also optimizes cache reuse:

Input	OpenCV ms	G-API/Fluid ms	Factor Times
320 x 240	1.16	0.53	2.17x
640 x 480	5.66	1.89	2.99x
1280 x 720	17.24	5.26	3.28x
1920 x 1080	39.04	12.29	3.18x
3840 x 2160	219.57	51.22	4.29x

- The more data is processed, the bigger "G-Effect" is.

Understanding the "G-Effect"

Relative speed-up based on cache efficiency



The higher resolution is, the higher relative speed-up is (with speed-up on QVGA taken as 1.0).

Resources on G-API

Resources on G-API

Repository

- https://github.com/opencv/opencv (see modules/gapi)

Article

- https://opencv.org/hybrid-cv-dl-pipelines-with-opencv-4-4-g-api/

Documentation

- https://docs.opencv.org/4.4.0/d0/d1e/gapi.html

Tutorials

- https://docs.opencv.org/4.4.0/df/d7e/tutorial_table_of_content_gapi.html

Thank you!