PyVX Documentation

Release 0.2.7

Hakan Ardo

December 08, 2014

Contents

PyVX is an implementation of OpenVX in python. OpenVX is a standard for expressing computer vision processing algorithms as a graph of function nodes. This graph is verified once and can then be processed (executed) multiple times. This implementation gains its performance by generating C-code during the verification phase. This code is compiled and loaded dynamically and then called during the process phase.

To use this python implementation as an OpenVX backend from a C program, a shared library is provided. This library embeds python and provides an C API following the OpenVX specification. That way the C program does not need to be aware of the fact that python is used. Also, any C program following the OpenVX specification will be compilable with this backend.

CHAPTER 1

Status

This is currently only a prof of concept. Most of the OpenVX functionality is still missing. Some small examples are working. See the demo directory. A handful of nodes are implemented as well as graph optimizations to do dead code elimination and to merge strictly element-wise nodes. Contributions are welcome.

CHAPTER 2

Installation

Before installing, make sure all dependencies are installed (the package will install anyway, but some functionality will be missing):

```
apt-get install vlc libvlc-dev freeglut3-dev
```

Then there are a few different ways to install PyVX:

- Use pip:

```
pip install pyvx
```

- or get the source code via the Python Package Index.
- or get it from Github:

```
git clone https://github.com/hakanardo/pyvx.git
cd pyvx
python setup.py install
```

This will install the backend and the python API. If you also want the C API which allows you to compile OpenVX programs written in C and have them use PyVx as their backend you also need to:

```
sudo python -mpyvx.capi build /usr/local/
```

This will install *libopenvx.so** into */usr/local/lib* and place the OpenVX headers in */usr/local/include/VX*.

CHAPTER 3

Modules

The main modules of PyVX are:

pyvx.vx C-like Python API following the OpenVX API as strictly as possible.

pyvx.pythonic A more python friendly version of the OpenVX API.

pyvx.capi A specification of a C API that is used generate a shared library and a header file that embeds python and calls the `pyvx.vx` functions. This aims to provide the C API as it is specified by the OpenVX standard.

pyvx.nodes The implementation of the different processing nodes.

pyvx.optimize Graph optimizations that are executed on the graphs during the verification step.

pyvx.codegen Code generation tools.

3.1 pyvx.vx — C-like Python API

This module provides the functions specified by the OpenVX standard. Please refer to the OpenVX speficication for a description of the API. The module name vx is used instead of a vx prefix on all symbols. The initial example on page 12 of the specification would in python look like this:

```
from pyvx import vx

context = vx.CreateContext()
images = [
    vx.CreateImage(context, 640, 480, vx.DF_IMAGE_UYVY),
    vx.CreateImage(context, 640, 480, vx.DF_IMAGE_U8),
    vx.CreateImage(context, 640, 480, vx.DF_IMAGE_U8),
]
graph = vx.CreateGraph(context)
virts = [
    vx.CreateVirtualImage(graph, 0, 0, vx.DF_IMAGE_VIRT),
    vx.CreateVirtualImage(graph, 0, 0, vx.DF_IMAGE_VIRT),
    vx.CreateVirtualImage(graph, 0, 0, vx.DF_IMAGE_VIRT),
    vx.CreateVirtualImage(graph, 0, 0, vx.DF_IMAGE_VIRT),
]
vx.ChannelExtractNode(graph, images[0], vx.CHANNEL_Y, virts[0])
vx.Gaussian3x3Node(graph, virts[0], virts[1])
vx.Sobel3x3Node(graph, virts[1], virts[2], virts[3])
vx.MagnitudeNode(graph, virts[2], virts[3], images[1])
vx.PhaseNode(graph, virts[2], virts[3], images[2])
status = vx.VerifyGraph(graph)
if status == vx.SUCCESS:
```

```
    status = vx.ProcessGraph(graph)
else:
    print("Verification failed.")
vx.ReleaseContext(context)
```

3.2 pyvx.pythonic — Python friendly API

Here a transformed OpenVX API is provided that is intended to feel more natural to a python programmer. The above example would with this API look like this:

```
from pyvx import *

with Graph() as g:
    img = Image(640, 480, DF_IMAGE_UYVY)
    smooth = Gaussian3x3(img.channel_y)
    dx, dy = Sobel3x3(smooth)
    mag = Magnitude(dx, dy)
    phi = Phase(dx, dy)
    mag.force()
    phi.force()
g.verify()
g.process()
```

This API is generated from the OpenVX API using the following transformations:

- The error codes `VX_ERROR_XXX_YYY` are turned into exceptions `XxxYyyError` and raised instead of returned.
- Graph's are created using

  ```
  class Graph(context=None, early_verify=True):
  ```

  If `context` is not specified a single global context will be created an used. If `early_verify` is `True` a partial verification will be performed as the nodes are created. This allows most errors to be detected at this time and raised as exceptions. The tracebacks of those exceptions will point to the line producing the erroneous node. This simplifies debugging a lot.

  The `Graph` objects are context manager that support the `with` statement as shown in the example above. It is used to make all the nodes and virtual images produced from within the code block belong to that graph. This allows virtual images to be automatically created and the use of special methods to create binary operations (see below).
- Within the code block of a `with Graph():` construction, the following features can be used:
  - For each `vxXxxNode` there is a `Xxx()` function that has only the input arguments of `vxXxxNode`. This function will create a 0x0 virtual image with color `DF_IMAGE_VIRT` for each of the output and inout arguments. Then it will call `vxXxxNode` and return the created images. Also, most non-image input arguments have been given default values and can be skipped.
  - The `Image` objects have `width`, `height` and `color` properties than can be adjusted at any time before the verification phase. However, to get the most out of the early verification described above, it is recommended to make any such adjustments as soon as possible.
  - `Image` objects have a `force()` method that will turn a virtual image into a normal fully allocated non-virtual image.

- `Image` objects that have been passed as an output or inout parameter of a node have a `producer` property that refers to this node. It can be used to access a node object even if only it's outputs are available (which would be the typical case when using this API).
- For each `VX_CHANNEL_X` the `Image` object has a `channel_x` property that will create a `ChannelExtractNode` and return it's virtual output images.
- A lot of the python special methods are Implemented on the `Image` objects to allow numpy style expressions to be used to create nodes.

As an example here is a graph that calculates the squared magnitude in a 32 bit unsigned image:

```
from pyvx import *

with Graph() as g:
    img = Image(640, 480, DF_IMAGE_UYVY)
    dx, dy = Sobel3x3(img.channel_y)
    mag = dx*dx + dy*dy
    mag.color = DF_IMAGE_U32
    mag.force()
g.verify()
g.process()
```

3.3 `pyvx.capi` — C API

This module allows the use of this python implementation as an OpenVX backend from a C program. A shared library is provided that embeds python and exports a C API following the OpenVX specification. That way the C program does not need to be aware of the fact that python is used. Also, any C program following the OpenVX specification should be compilable with this backend.

```
sudo python -mpyvx.capi build /usr/local/
```

This will install *libopenvx.so** into */usr/local/lib* and place the OpenVX headers in */usr/local/include/VX*.

3.4 `pyvx.nodes` — Node implementations

This module contains the implementations of the different processing nodes. They are implemented by subclassing `Node` and overriding `signature`, `verify()` and `compile()`. As an example here is the implementation of the Gaussian3x3Node:

```
class Gaussian3x3Node(Node):
    signature = "in input, out output"

    def verify(self):
        self.ensure(self.input.image_format.items == 1)
        self.output.ensure_similar(self.input)

    def compile(self, code):
        code.add_block(self, """
            for (long y = 0; y < img.height; y++) {
                for (long x = 0; x < img.width; x++) {
                    res[x, y] = (1*img[x-1, y-1] + 2*img[x, y-1] + 1*img[x+1, y-1] +
                                 2*img[x-1, y]   + 4*img[x, y]   + 2*img[x+1, y]   +
                                 1*img[x-1, y+1] + 2*img[x, y+1] + 1*img[x+1, y+1]) / 16;
                }
```

```
        }
        """, img=self.input, res=self.output)
```

- `Node.signature` is a string specifying the argument names and there directions (in, out or inout). The arguments will be assigned to attributes with the same names when the node is created. The arguments can be given default values by assigning them to class-level attributes.
- `Node.verify(self)` is called during the verification phase and can assume that all nodes it depend on have verified successfully. It is supposed to check the arguments and raise one of the `VerificationError`'s if they don't make sense. Also any output images with width/height set to 0 or color set to DF_IMAGE_VIRT should be given proper values. There are a few helper methods available described below.
- `Node.compile(self, code)` is called after verification of the entire graph was successful. It is responsible for generating C code implementing the node using the `code` argument. It has a notion of magic variables used to abstract away the pixel access calculations. See `pyvx.codegen.Code`.

To simplify the implementation of `verify()` there are a few helper functions. They will updated the properties of the images if they've not yet been set, and raise `ERROR_INVALID_FORMAT` if they were set to something different.

- `Image.ensure_shape(self, other_image)` Ensures `self` has the same width and height as other_image.
- `Image.ensure_shape(self, width, height)` Ensures `self` has the width `width` an the height `height`.
- `Image.ensure_color(self, color)` Ensures that the color of `self` is `color`
- `Image.suggest_color(self, color)` Sets `self.color` to `color` if it is not yet specified.
- `Image.ensure_similar(self, image)` Ensures that the shape and number of channels of `self` and `image` are the same, and suggests that the color of `self` is the same as `image`.
- `Node.ensure(self, condition)` raises `ERROR_INVALID_FORMAT` if `condition` is `false`.

To allow for a general implementation of the graph optimizations, there are special subclasses of `Node` that should be used. For strictly element-wise operations there is `ElementwiseNode`. It expects a `body` attribute with the code that will be executed for each pixel. This code can assume that there are variables with the same name as the arguments available. For the input arguments those variables contains the pixel values of the current pixel and it is the responsibility of this code to assign the output variables. As an example, here is the implementation of the `PowerNode`:

```
class PowerNode(ElementwiseNode):
    signature = "in in1, in in2, in convert_policy, out out"
    body = "out = pow(in1, in2);"
```

If some logic is needed to produce the code, `body` can be implemented as a `property`. That is for example done by the `BinaryOperationNode`:

```
class BinaryOperationNode(ElementwiseNode):
    signature = "in in1, in op, in in2, in convert_policy, out out"
    @property
    def body(self):
        return "out = in1 %s in2;" % self.op
```

The default `ElementwiseNode.verify()` will ensure that the output images are of the same size as the input images and will use `pyvx.types.result_color()` to replace `DF_IMAGE_VIRT` colors among the output images. If this is not appropriate it can be overridden.

3.5 pyvx.codegen — Code generation tools

class pyvx.codegen.**Code**(*code=''*)

Represents some generated C-code together with a bit of metadata. It has the following public attributes:

indent_level Number of spaces to indent code added using add_block.

extra_link_args A list of extra arguments needed to be passed to the linker when compiling the code. It is typically used to link with external libraries used by the code.

includes A set of lines added at the top of the generated .c file outside the function enclosing the code. This is intended for #include ... lines.

add_block(*cxnode*, *code*, ***magic_vars*)

Append code as a new block of code. It will be enclosed in with {} brackets to allow it to declare local variables. The code will be parsed and all references to the symbol names passed as keyword arguments will be extracted and handled separately. These magic variables are intended to refere to Image objects, but could be anything that define compatible getattr() and getitem() methods. If an Image is passed as the keyword argument img, it can be used in the C-code in the following ways:

img[x,y] The value of pixel (x, y) of a single channel image.

img.channel_x[x,y] The value of pixel (x, y) in CHANNEL_X.

img[i] The i'th value in the image. i is an integer between 0 and width * height * channels - 1.

img.channel_x[i] The i'th value in CAHNNEL_X of the image. i is an integer between 0 and width * height - 1.

img.width The width of the image in pixels.

img.height The height of the image in pixels.

img.pixels The number of pixels in the image (width * height).

img.values The number of values in the image (width * height * channels).

img.data A pointer to the beginning of the pixel data.

add_code(*code*)

Extend the code with code without any adjustment.

CHAPTER 4

Comments and bugs

There is a mailing list for general discussions and an issue tracker for reporting bugs and a continuous integration service that's running tests.

p

A

C

P