On the Images Pipe Line Filtering

Dmitry V. Yurin

Moscow State University, Faculty of Computational Mathematics and Cybernetics, Lab. of Mathematical Methods of Image Processing

yurin_d@inbox.ru

A lot of filtering schemes used in image processing can be adequately presented as a directed graph of elementary filters such as convolutions by row and columns, pixel-wise operations (summation, function calculation), range filtering. A typical example is Canny edge detection (Figure 1.), which consists of convolution with Gaussians and its derivative, computation of gradient vector module and detection of its maxima (by 3x3 environment). Its straightforward numerical implementation requires at least 2 intermediate images allocated in memory: two inputs and one output of module calculation block, the output can be shared with one of the inputs. For more complex filters, such as color edge detection (Di Zenzo 86), Harris corner detector and its color version, etc., the number of such required intermediate images can be much greater. It should be mentioned that it is more suitable to perform some operations in floating point so additional data containers are required for intermediate steps. Currently typical images sizes are grow significantly and these large memory requirements become unacceptable.

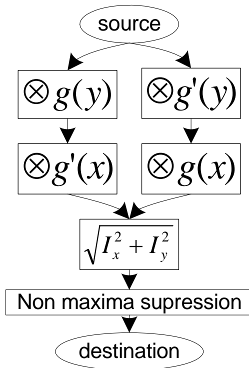

Figure 1. Canny edge detector

This paper is concentrated on the class of filers, for each block of the filters it is true: output pixel value at (x,y) coordinates depends only on input images pixels at the same relative coordinates and a small environment of it. Note that for this class of filters it is not necessary to hold in memory whole intermediate images, but for processing each image row it is required only a small strip of corresponding rows of the filter input. Thus, complex filters like in Figure 1 can be implemented in a pipeline style. Another issue is that of such code must be implemented once in generalized library and used for all filters implementation. The library usage must be easy – the image processing goal is image processing, not memory distribution!

The C++ template library proposed solves this problem. Typical program code contains the following mandatory fragments:

1) elementary filters creation like:

```
Filter1<float> f1(..filter parameters..);
```

2) filters connection directly reflects its graph (like in Figure 1):

```
f1.out(2).ConnectTo(&f3.in(0));
```

3) system engine creation, initialization and run :

```
FiltersSystem sys("test.dot");
sys.Assign(&src,&dst);
sys.Run();
```

At the Assign() step the library can inform the user (if indicated) what filters and ports are badly connected and refer to the saved graph (debugging) in GraphViz (www.**graphviz**.org) format.

To create an elementary filter it is only required to request the desired environment size (specified by struct Env) and overload virtual function ProcessLine() of the base class FilterElementary. Each elementary filer have a set of inputs and outputs. Each output encapsulates StripBuffer of the size sufficient to fulfill requests from all inputs connected to this output. Source and target filters contain no inputs or outputs, correspondingly.

It is important that each StripBuffer containing more than 1 image row results in delay in pipelining. The buffers sizes and accordingly delays depend on filter parameters and subjected to changes. On the other hand some elementary filters have more than 1 input and for the filters to work properly data on their inputs must be synchronized, that is all buffers on inputs must contain the required strip around the same row of input images. The library proposed performs this synchronization automatically by increasing delays via increasing `Env`.f for some branches in the complex filter's directed acyclic graph. For this purpose the initial buffer assignment algorithm from [1] primary proposed for chips development have been adopted:

```
1  foreach u ∈ V  in TS order
2    if  u ∈ S  d[u] ← 0;
3    else calcDelay(u)
4  foreach u ∈ V  in TS order
5    initStripBuffers(u)
```

here function **`calcDelay(u)`** is:

```
1  foreach  v ∈ u.{In}
2   w ← d[v]+u.In.Env.f/v.Out.H
3   d[u] ← max(d[u],w)
4  foreach  v ∈ u.{In}
5   w ← u.In.Env.f/v.Out.H
6   b[(v,u.In)] ← d[u]-d[v]-w
```

and function **`initStripBuffers(u)`** is

```
1  env ← {0,0,0,0}
2  foreach  v.In ∈ u.{Out}
3   en ← v.In.Env
4   f← u.Out.H*b[(u,v.In)]
5   en.f← round(en.f+f+0.5)
6   env ← max(env,en);
7   u.Out.initStripBuffer(env);
```

The library makes filters development fast and easy and ensures large memory saving at the cost of small performance losses.

The research is done under support of RFBR, grants 06-01-00789-a, 08-07-00469-a.

[1] Chatterjee M., Banerjee S., Pradhan D.,K., 2000. Buffer Assignment Algorithms on Data Driven ASICs //In IEEE Trans. on computers. V. 49, No.1. P.16—32.