Hardware image filtering on desktop computers

Janek Press
Institute of Computer Science, Faculty of Mathematics and Information Science
University of Tartu, Tartu, Estonia
press@ut.ee

Abstract

Real-time video processing, large scale image manipulation and many other fields require much processing time. Severity of this problem can be abated in different ways. One solution is to use what is already present in a desktop personal computer as general purpose graphics hardware gets more programmable.

This presentation gives an overview of author's experiences and work in this field. Test results are promising: hardware filters are up to 32 times faster in perfect conditions and at least 3 times faster in common test. The following filters are tested: Gaussian and other smoothing techniques, sharpen, Roberts, Sobel, Prewitt, median, morphological corner extractor.

Keywords: *hardware, image filters.*

1. INTRODUCTION

Real-time and large scale image processing at high rates is computationally expensive. There are several approaches to gain speed: optimize code, use dedicated hardware, and revaluate methods used and goals to achieve.

As more powerful and programmable graphics hardware becomes available to mainstream consumer new approaches are applicable. On the other hand this process is accelerating and because of that some methods easy to accomplish using latest hardware can be at least partially approximated using already widely available devices. To explore the possibilities filters are developed and their usability tested.

First section of this paper gives some hints implementing spatial domain image filters in limited hardware support environment.

Second section describes a test system developed by the author for estimating speed gains resulting from moving calculations from CPU to GPU.

The reader is expected to have basic knowledge of shading languages. If not marked otherwise all calculations in hardware are performed between vectors. All 4 components are 12 bit fixed point real numbers with values ranging from -1 to 1 for signed and from 0 to 1 for unsigned format.

2. IMPLEMENTING FILTERS

In this paragraph a short introduction to writing filters for limited hardware is given.

2.1 Roberts

Common way to calculate any gradient filter is to assume $|a-b| \approx \sqrt{(a^2+b^2)}$. As this is equal to

$$-\left(\left(\left(-a\right)+b\right)+\left(a+\left(-b\right)\right)\right)$$

where a and b are unsigned and "-" is operator for unsigned invert. The proof of correctness is given by the author [6]. It is possible to calculate this NV10 class hardware. Register Combiner program implementing this is given on Figure 1, assuming that tex0 is a and tex1 is b.

```
!!RC1.0
{
  rgb
  {
    discard = unsigned(tex0);
    discard = unsigned_invert(tex1);
    spare0 = sum();
  }
}
{
  rgb
  {
    discard = unsigned_invert(tex0);
    discard = unsigned(tex1);
    spare1 = sum();
  }
}
out.rgb = unsigned_invert(spare0)+
          unsigned_invert(spare1);
out.a = unsigned_invert(zero);
```

Figure 1: Absolute difference between `tex0` and `tex1`.

For example using multiple passes and linear texture filtering NV10 architecture hardware requires 4 passes to implement Sobel filter but NV20 can do the same with 2 passes. Using NV30 just one pass is required.

2.2 Morphology

Morphological feature extractors in common [8] are multipass techniques. Such a filter needs to calculate two sums of some samples one of which should be zero and other should be one to pass the test. This can easily be achieved using PixelShader 1.4 and later versions but implementing this in version 1.1 is not straight forward.

If a test passes the output has one value (`c0`), otherwise another (`c1`). It is noticeable that actual minimal difference of importance in color values m (0.012) is much higher than the theoretical value (1/256 = 0.004) according to test on a GeForce3. Pixel Shader for this is on Figure 2.

```
ps_1_1
def c2, 0, 0, 0, 0
def c3, 0.488, 0.488, 0.488, 0.488
def c4, 1, 1, 1, 1
def c5, 0.162, 0.285, 0.067, 0

tex t0
tex t1
tex t2
tex t3

add_d2 r0, t0, t1
dp3 r0, r0, c5
cnd r1, r0.a, c3, c2

add_x4 r0, t2, t3
dp3_x4 r0, r0, c4
cnd r0, r0.a, c2, c3

add r0, r0, r1
cnd r0, r0.a, c0, c1
```

Figure 2: A Morphology pixel shader with 12 instructions. `c5` is half of RGB to intensity factor's components plus *m* and 0.01 to compensate for loss of precision.

The result of such a test does not change if the sum of samples with mask 0 is subtracted from the sum of samples with mask 1 and just one conditional test is done. Updated program is given on Figure 3.

```
ps_1_1
def c2, 1, 1, 1, 1
def c3, 0.162, 0.285, 0.067, 0

tex t0
tex t1
tex t2
tex t3

add_d2 r0, t0, t1
dp3 r0, r0, c3

add_x4 r1, t2, t3
dp3_x4 r1, r1, c2

add r0, r0, -r1
cnd r0, r0.a, c0, c1
```

Figure 3: Morphology pixel shader with 10 instructions.

2.3 Gaussian blur

As Gaussian filter is separable, so it can be achieved using a multipass approach: two passes using blend for two perpendicular directions.

For example having 4 texture units acceptable results can be achieved for standard deviation of 3. Already at this level, a compensation for loss of brightness must be added: sum of weight of samples not taken into account (farther than 4 texels) distributed to sampled texels.

But as hardware supporting 4 texture units is limited to 12 bit precision, larger standard deviation values would result texel weights less than *m*.

Examples using DirectX 9 class hardware are available by I. Takashi [3] and Frank Jargstorff [2].

2.4 Median filters

In DirectX 8 class even a 3 sample full color median filter can't be implemented correctly. Using PixelShader 2.0 only a 4 sample median filter can be implemented because the instruction count limit.

A 5 sample OpenGL fragment program is implemented by I. Takashi [4].

3. TEST SYSTEM

To compare speed and quality of different filters an extendable test suit was created by the author [6]. The second version of this software is available for free download as part of author's bachelor theses project [5]. The package contains besides the testing system all filters mentioned above as well some more. Also a sample framework for writing libraries is included.

The system consists of three parts:

- **Frontend** – convenient environment to control, test and compare filters.
- **Filters** – libraries implementing image filters: software, DirectX, OpenGL.
- **Inputs** – libraries for obtaining images to process. Besides JPEG, BMP and TGA formats AVI files and live video capturing is supported.

3.1 Writing filters

Basic filters to be tested can be hardcoded into filter libraries or be read from external files. For DirectX the effect file format (*.fx) is used, for OpenGL filters a simple script language was considered to be more appropriate (*.flt) as ARB fragment programs are not supported NV20 hardware.

Available data to a filter:

- one texture
  - `tex0` to `tex7` (OpenGL)
  - `InputTexture` (DirectX)
- one set of texture coordinates
- `texelSize` (4 componetnt vector):
  - texel width
  - texel height
  - half of texel width
  - half of texel height
- `factor` (4 componetnt vector, free usage)
- (OpenGL only) `ModelViewMatrix`
- (DirectX only) `PS`*`n`* and `VP`*`m`* pixel shader and vertex shader external reference (n and m are natural numbers)

For general efficiency [1] as for interesting techniques [7] two rules are common: keep it short and what could be done in a vertex shader must **not** be done in a pixel shader.

3.2 Combining filters

Filter libraries use special script file format to combine basic filters whit each other. These script files have same structure but different extensions because the source filters to be combined are designed either for DirectX or OpenGL (*.flX *.flG respectively).

3.2.1 File structure

File begins with an optional name of filter. Next one ore more lines are source source definitions in the following form:

```
<id>=<source file>
```

After that comes definition of the filter at one line. For the syntax of this expression refer to Figure 4. Lines beginning with "#" are comments.

```
<filter> -> <ident>
<filter> -> <op>(<param>,<param>)
<op>     -> x
<op>     -> *
<op>     -> +
<op>     -> ^
<param>  -> <op>
<param>  -> <id>
<param>  -> <id>{r}
<param>  -> <id>{r,r,r,r}
```

Figure 4: Set of productions defining syntax of filters definitions. r is a real number between 0 and 1 in it's decimal presentation.

3.2.2 Operators

Following binary operations are supported:

- **Composition** "x" - result of right filter is passed for input to left filter.
- **Multiplication** "*" - results of two filters are multiplied.
- **Addition** "+" - results of two filters are added.
- **Rise to power "^"** - shorthand to write several multiplications in compact form. Takes an integer for the right parameter.

3.2.3 Example filter

For illustration consider following problem: replace pixels in source image if these classify as corners in morphological filter of different types.

Assume we have identity filter, classification filter that outputs one, if source texel is less bright than a constant, and morphological corner extractor based on ideas of Section 2.2. To compute desired effect one applies the morphological filter which paints all corners in different colors and classifies these as texels to be replaced in the original image. Now, according to this classification texels are added to output image so that colors of corner texels are replaced according to classification: red is upper left, white is upper right, green is bottom left and blue is bottom right. Script for this composition is on Figure 5.

```
Added morph

morph=DX8_morphology.fx
i=DX_identity.fx
cln=DX8_classify_neg.fx

+(morph,*(i,x(cln{1,1,1,0.1},morph)))
```

Figure 5: Composite filter adding results of morphological tests to original image.

3.3 Test results

For basic testing a system with 1.4GHz processor and a GeForce 3 graphics board was used. Advanced filters were tested on machine with a 2.8GHz Pentium 4 and GeForceFX 5950 Ultra. Using Radeon 8xxxx and 9xxx series cards lead to a performance drop at huge inputs.

3.3.1 Video capturing

Standard webcam delivers about 6 frames per second with a resolution of 320x240. These frames were passed to edge extracting filters in real-time and the CPU usage was observed.

Applying hardware identity filter compared to a software filter raised the CPU load about 15%. This is because of the time needed to transfer the data to and from the GPU.

Using Roberts filter gives a slight advantage but not more than 5% compared to a software filter.

Sobel filter consumed all available CPU time in software mode but did not arise the load noticeable compared to level of applying hardware Roberts filter.

There was no practical advantage using GPU based filters for the simplest cases but already Sobel filter in hardware reduced the CPU load approximately **3 times**.

3.3.2 Static images

As mentioned before transfer times to and from the graphics hardware must be taken into account. To illustrate this image of size 512x512 was used. As it was the case with video processing using hardware implementation for Roberts filter lead to a loss of performance (25%) and Sobel filters had opposite results (2.5 times faster). For larger scale images (1024x1024) the hardware assisted version of Roberts filter performs better giving positive results and Sobel filter extends its lead reducing to CPU load about **4 times**.

3.3.3 Pure tests

To estimate full potential of more complex hardware filters the system provides a mechanism to disable input and output for each time a filter is executed. Instead of this the data is transferred only at the beginning and at the end of a testing session. This is meaningful because during execution of complex filters the data does not leave the video memory.

Using this approach this particular GPU outperforms the CPU at least **32 times**.

4. CONCLUSION

GPUs get every year more programmable gaining new functionality and speed. To have the greatest benefit through knowing your limits these must be thoroughly explored.

First of all some filters of interest can be effectively implemented already using DirectX 8 class hardware and practical advantage is noticeable. Secondly most of them are multipass. Last but not least after working for several hours on a filter just one instruction too much from making it work you become aware where the limit is so one should not forget about the hardware already widespread and techniques used in the past.

5. REFERENCES

[1] A. Rege, C. Brewer. Practical performance analysis and tuning.

[2] F. Jargstorff. *Gpu image processing: The cg_scotopic demo.*

[3] I. Takashi. Gaussian filter.

[4] I. Takashi. Median filter.

[5] J. Press Hardware image filtering (bachelor thesis).

[6] J. Press. Hardware image filtering (semester work).

[7] M. M. Wloka. Gpu-assisted assisted rendering techniques.

[8] R Fisher, S. Perkins, A. Walker, E. Wolfart. *Hipr2*.

About the author

Janek Press is graduate student at University of Tartu, Faculty of Mathematics and Information Science His contact email is press@ut.ee.