FPGA-based Acceleration for various Image Processing Techniques

Aryan Agarwal
Electronics and Communication Engineering
2018102024

Utkarsh Mishra
Electronics and Communication Engineering
2018102020

***Abstract*—We have tried to accelerate various image filtering techniques on AWS EC2 F1.2x large instance in this project. We started with a basic Mean filter to get baseline performance, then moved onto further complex filters like Emboss filter and Unsharp filter and finally the LoG filter. We have tried various optimisation techniques to get better performance.**

I. Introduction

We have chosen 15×15 filters, which are quite large. The reason for this is to reduce noise and get more smooth images for further operations.

A. Mean filter

A box blur is a spatial domain linear filter in which each pixel in the resulting image has a value equal to the average value of its neighboring pixels in the input image. Due to its property of using equal weights, it is one of the fastest among other filters for convolution. It is a form of low-pass ("blurring") filter. The filter that we have used is:

$$M = \frac{1}{225}\begin{pmatrix} 1 & 1 & \dots \\ \vdots & \ddots & \\ 1 & & 1 \end{pmatrix}_{15\times 15}$$

Box blurs are frequently used to approximate a Gaussian blur. By the central limit theorem, repeated application of a box blur will approximate a Gaussian blur.

Original Image

Image with Box Blur

B. Emboss Filter

The Emboss filter, also called a directional difference filter, enhances edges in the direction of the selected convolution mask(s). When the emboss filter is applied, the filter matrix is in convolution calculation with the same square area on the original image. So it involves a large amount of calculation when either the image size or the emboss filter mask dimension is large. The emboss filter repeats the calculation as encoded in the filter matrix for every pixel in the image; the procedure itself compares the neighboring pixels on the image, leaving a mark where a sharp change in pixel value is detected. In this way, the marks form a line following an object's contour. The process yields an embossed image with edges highlighted. Four primary emboss filter masks are:

$$\begin{pmatrix} 0 & +1 & 0 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} +1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$
$$\begin{pmatrix} 0 & 0 & 0 \\ +1 & 0 & -1 \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & +1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

The filter that we have used is:

$$M = \frac{1}{225}\begin{pmatrix} -1 & -1 & \dots & -1 & 0 \\ -1 & \dots & \dots & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & 0 & \dots & 1 & 1 \\ 0 & 1 & \dots & 1 & 1 \end{pmatrix}_{15\times 15}$$

Example: Two different emboss filters are applied to the original photo. Image (a) is the result of a 5×5 filter with the +1 and -1 in the horizontal direction, which emphasizes vertical lines. Image (b) is the result of a 5×5 filter with the +1 and -1 in the vertical direction; it emphasizes horizontal lines. Since the entries of a given emboss filter matrix sum to zero, the output image has an almost completely black background, with only the edges visible.

C. Unsharp Filter

Unsharp masking (USM) is an image sharpening technique, first implemented in darkroom photography, but now commonly used in digital image processing software. Its name derives from the fact that the technique uses a blurred, or "unsharp", negative image to create a mask of the original

image. The unsharp mask is then combined with the original positive image, creating an image that is less blurry than the original. The resulting image, although clearer, may be a less accurate representation of the image's subject.

Example: In the example below, the image is convolved with the following sharpening filter:

$$M = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

This matrix is obtained by taking the identity kernel and subtracting an edge detection kernel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

The sharpening effect can be controlled by varying the contribution of edge detection.

Fig. 1. Sharpened twice

The filter that we have used is:

$$M = \frac{1}{225} \begin{pmatrix} -1 & -1 & \dots & \dots & \dots & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & \dots & -1 & 224 & -1 & \dots & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -1 & -1 & \dots & \dots & \dots & -1 & -1 \end{pmatrix}_{15\times 15}$$

D. Laplace of Gaussian (LoG) Filter

Laplacian filters are derivative filters used to find areas of rapid change (edges) in images. Since derivative filters are very sensitive to noise, it is common to smooth the image (e.g., using a Gaussian filter) before applying the Laplacian. This two-step process is call the Laplacian of Gaussian (LoG) operation.

$$L(x,y) = \nabla^2 f(x,y) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

There are different ways to find an approximate discrete convolution kernal that approximates the effect of the Laplacian. A possible kernel is:

$$M = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

This is called a negative Laplacian because the central peak is negative. It is just as appropriate to reverse the signs of the elements, using -1s and a +4, to get a positive Laplacian. It doesn't matter.

To include a smoothing Gaussian filter, combine the Laplacian and Gaussian functions to obtain a single equation:

$$LoG(x,y) = -\frac{1}{\pi\sigma^4}\left[1 - \frac{x^2+y^2}{2\sigma^2}\right] e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The LoG operator takes the second derivative of the image. Where the image is basically uniform, the LoG will give zero. Wherever a change occurs, the LoG will give a positive response on the darker side and a negative response on the lighter side. At a sharp edge between two regions, the response will be:

- zero away from the edge
- positive just to one side
- negative just to the other side
- zero at some point in between on the edge itself

Fig. 2. Original Image

Fig. 3. LoG Filtered Image

The filter that we have used is:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	2	3	4	3	2	0	0	0	0	0
0	0	0	0	3	6	10	10	10	6	3	0	0	0	0
0	0	0	2	6	11	6	0	6	11	6	2	0	0	0
0	0	0	3	10	6	-25	-48	-25	6	10	3	0	0	0
0	0	0	4	10	0	-48	-83	-48	0	10	4	0	0	0
0	0	0	3	10	6	-25	-48	-25	6	10	3	0	0	0
0	0	0	2	6	11	6	0	6	11	6	2	0	0	0
0	0	0	0	3	6	10	10	10	6	3	0	0	0	0
0	0	0	0	0	2	3	4	3	2	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

II. Optimisations

We did the following optimisations in the convolution code on the kernel:

- `#pragma HLS PIPELINE II=1`
  By adding this line, the pragma tells the compiler that a new iteration of the for-loop should be started exactly one clock cycle after the previous one. As a result, the SDAccel compiler builds an accelerator with 15*15=225 multipliers. Normally the 225 multiply-add operations of the 2D convolution would be executed sequentially on a conventional CPU architecture, they are executed in parallel in the purpose-built FPGA accelerator. This results in a significant performance improvement.

- Usage of sliding-window approach to get the correct window pixels of the image. This reduces a lot of communication cost.

- Allocated contiguous memory to input image, so that streaming pixels is easier.

We also experimented with the number of kernels to get better performance. We have used picadilly.bmp as the input and ran the code for 10 times on the same input image (the results are for 10 runs).

A. 1 kernel

1) Unsharp filter:

- FPGA Time: 0.373306 s
- FPGA Throughput: 158.921 MB/s
- CPU Time: 8.58447 s
- CPU Throughput: 6.91087 MB/s
- FPGA Speedup: 22.9958 x

2) Mean filter:

- FPGA Time: 0.309095 s
- FPGA Throughput: 191.935 MB/s
- CPU Time: 8.7733 s
- CPU Throughput: 6.76212 MB/s
- FPGA Speedup: 28.3838 x

3) LoG filter:

- FPGA Time: 0.309322 s
- FPGA Throughput: 191.794 MB/s
- CPU Time: 17.5655 s
- CPU Throughput: 3.37743 MB/s
- FPGA Speedup: 56.787 x

4) Emboss filter:

- FPGA Time: 0.313735 s
- FPGA Throughput: 189.096 MB/s
- CPU Time: 8.6584 s
- CPU Throughput: 6.85186 MB/s
- FPGA Speedup: 27.5978 x

B. 3 kernels

1) Unsharp filter:

- FPGA Time: 0.192664 s
- FPGA Throughput: 307.926 MB/s
- CPU Time: 8.73027 s
- CPU Throughput: 6.79546 MB/s
- FPGA Speedup: 45.3135 x

2) Mean filter:

- FPGA Time: 0.133735 s
- FPGA Throughput: 443.61 MB/s
- CPU Time: 9.06806 s
- CPU Throughput: 6.54232 MB/s
- FPGA Speedup: 67.8062 x

3) LoG filter:

- FPGA Time: 0.130855 s
- FPGA Throughput: 453.374 MB/s
- CPU Time: 17.7707 s
- CPU Throughput: 3.33843 MB/s
- FPGA Speedup: 135.805 x

4) Emboss filter:

- FPGA Time: 0.13359 s
- FPGA Throughput: 444.092 MB/s
- CPU Time: 8.85563 s
- CPU Throughput: 6.69926 MB/s
- FPGA Speedup: 66.2897 x

C. 6 kernels

1) Unsharp filter:

- FPGA Time: 0.324142 s
- FPGA Throughput: 183.025 MB/s
- CPU Time: 8.83279 s
- CPU Throughput: 6.71659 MB/s
- FPGA Speedup: 27.2498 x

2) Mean filter:

- FPGA Time: 0.265641 s
- FPGA Throughput: 223.332 MB/s
- CPU Time: 8.89386 s
- CPU Throughput: 6.67046 MB/s
- FPGA Speedup: 33.4808 x

3) LoG filter:

- FPGA Time: 0.26145 s
- FPGA Throughput: 226.912 MB/s
- CPU Time: 17.6173 s
- CPU Throughput: 3.3675 MB/s
- FPGA Speedup: 67.3829 x

4) Emboss filter:

- FPGA Time: 0.268098 s
- FPGA Throughput: 221.285 MB/s
- CPU Time: 8.9658 s
- CPU Throughput: 6.61694 MB/s
- FPGA Speedup: 33.4423 x

We tried some more optimisations, but we were not successful implementing them:

- Pipelined kernel execution to minimise idle time for kernels.

III. Roofline Analysis

The peak compute performance for F1.2x large instance is

$$= \text{Number of DSPs} \times \text{Frequency(Gbps)}$$

$$= 6840 \times 16$$

$$= 1710\,\text{GFlops/s}$$

The DDR memory bandwidth is 16 Gbps. The number of computations in our code:

$$= 225 \times 2\ \text{(Convolution)} + 2\ \text{(Normalisation)}$$

$$= 452$$

We are using a sliding window approach for getting image values. The number of bytes required are:

$$= 225{\times}1\ \text{(U8 elements for filter)} + 15{\times}1\ \text{(U8 elements for Image)}$$

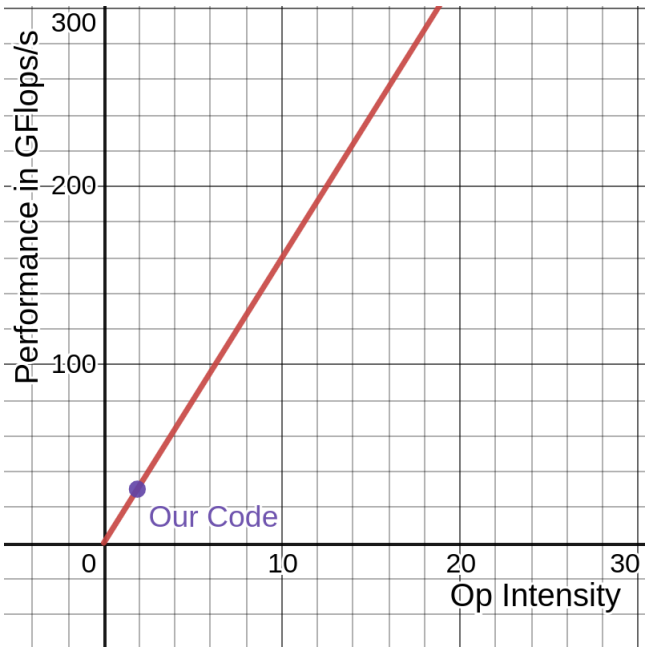
$$= 240$$

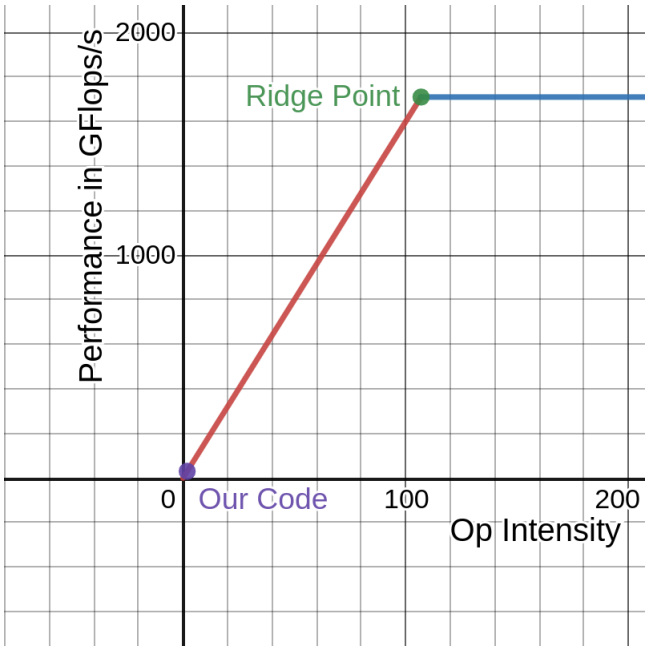
The operational intensity (OI) for our code:

$$= 452/240$$

$$= 1.883334$$

The ridge point is at OI = 106.875, but OI for our code is 1.883334, hence our code is memory bound. The peak attainable performance comes out to be 30.13334 GFlops/s.





IV. Results

We have attached the vitis analyser report that contains timing information, latency information and area information. We have also attached our source code. We used this code to generate the LoG filter. In the end, we saw using 3 kernels performed better than using 1 or 6 kernels. When using 1 kernel, the YUV-channel requests are sent sequentially. When using 3 or 6 kernels, the YUV-channel requests are sent in a parallelised way. We believe that using 6 kernels does not give the best performance because there are not enough resources on FPGA to manage 6 kernels. Also it may be possible that kernel switching is taking a lot of time.

Fig. 4. Input Image

Fig. 5. Outline Filter Output

Fig. 6. Mean Filter Output

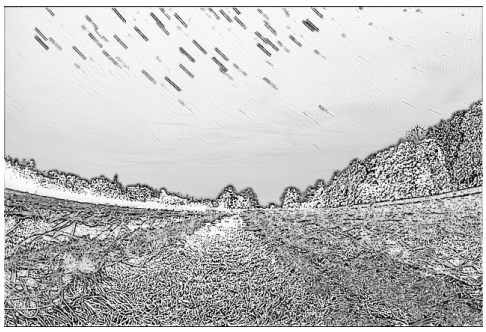


Fig. 7. LoG filter Output

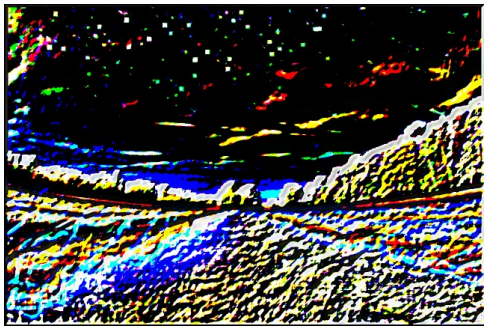


Fig. 8. Emboss Filter Output

Fig. 9. Input Image

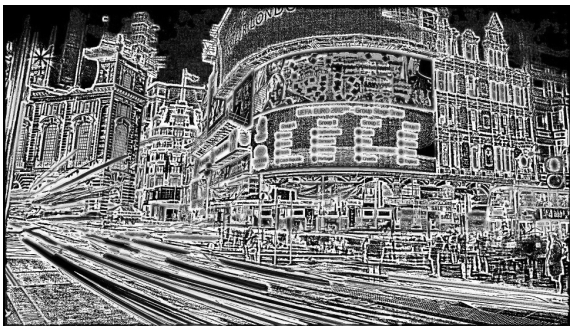


Fig. 10. Outline Filter Output

Fig. 11. Mean Filter Output

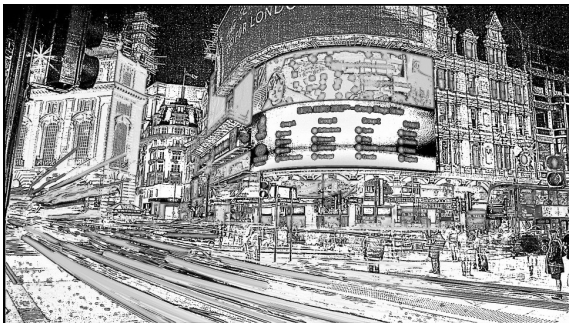


Fig. 12. LoG filter Output

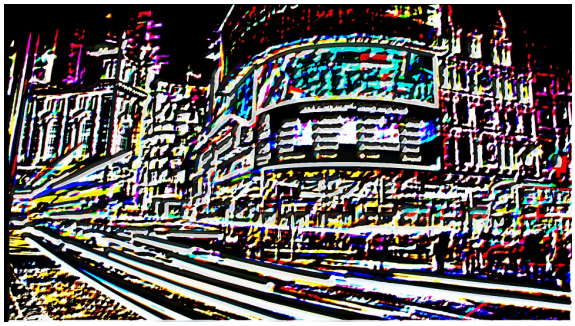


Fig. 13. Emboss Filter Output

REFERENCES

We referred to this tutorial to understand how to send images as a stream and receive them as a stream using buffers.