FPGA

SMOOTH FILTER ON FPGA

November 30, 2016

PORTO KATRINI CHARALAMPAKI
Master in Computer Vision
University of Burgundy

Contents

Chapter 1

Introduction

Nowadays, image processing tools are very easy to use and reviewed a lot. Applying a filter on an image is straight forward with a lot of programming languages thanks to well implemented libraries and toolboxes. However, some technologies can overtake the speed of such common tools by tuning the behavior of the hardware itself. The FPGA technology is one of those and can be described using the VHDL language. This allows to control any kind of logical gate that is needed to perform a proper filter for a real time processing application.

We detail in this report the different steps and components to design in order to achieve a fully tuned smooth filter with 3x3 neighbors, starting from the cache memory to the processing part. We also describe how to manage such a component and what are the results coming from these different managements.

Chapter 2

CACHE MEMORY

The cache memory is the part where the pixels will be stored in order to create a patch of size 3 by 3. As the pixels are stacked by row in a file, a solution has to be found to put aside the remaining pixels of the row, which are actually all the pixels except 3 of them.

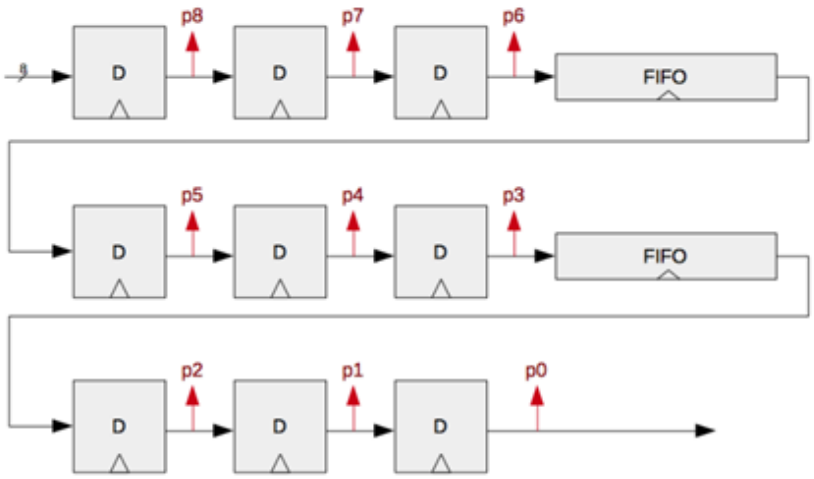

Figure 2.1: Cache memory framework to apply a 3x3 filter

One of the solutions is to read sequentially pixel by pixel the data of the file, make them pass through 3 simple flip-flops, before entering a fifo. The fifo is therefore keeping the data until it reaches a certain amount corresponding to the quantity of pixels needed to access a second row. This amount is the *PROG_FULL_THRESH* of the fifo. Once the quantity of data overpasses the threshold, an output called *PROG_FULL* send a positive signal.

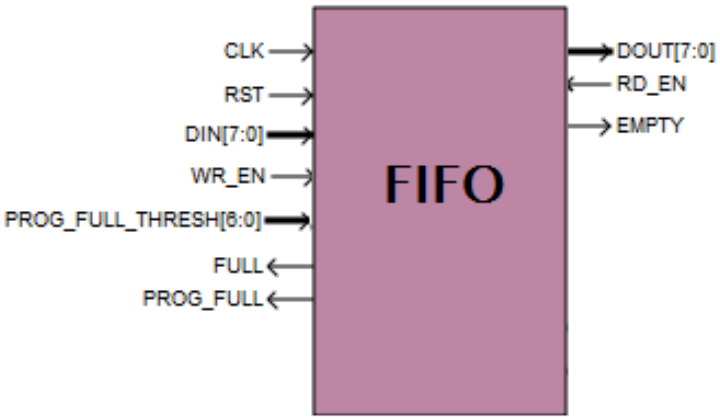

Figure 2.2: Designed fifo with inputs and outputs

Our idea is to connect this signal to the *RD_EN* pin of the fifo, in order to send data to the next level of flip-flops only after filling up the fifo with one image row. Like that the pixels passing through the next 3 flip-flops will correspond to the upper neighbors, in the image, of the pixels passing currently through the first flip-flops. We repeat the same process with an other fifo and three other flip-flops, what result in a filter of 3x3 neighbors.

The constant threshold entered inside the fifo corresponds to the number of pixels in a row, minus three pixels (the ones already in the flip-flops), minus the delay between the signal on the pin *PROG_FULL* and the output of the fifo.

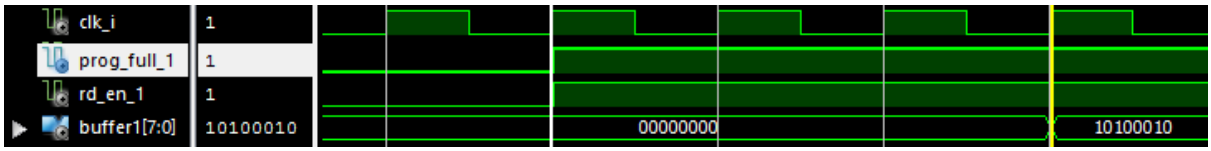

Figure 2.3: Smooth filter applied with undefined borders

In our example, we have an image of 128x128 pixels. So we fixed the threshold at $128-3-2=123$. As you can see in figure 2.3, the output *buffer1* of the fifo is only fed with data 2 clock cycles after the rising edge of the *prog_full_1*.

The *PROG_FULL_THRESH* pin needs only 7 bits to encode the number 123 and the *DIN* and *DOUT* pins embed pixels coded on 8 bits.

Chapter 3

PROCESSING

Once pixels are available in the cache memory, we can take the output of the 9 flip-flops as entries for our processing. A part of the RTL schematic of the cache memory is included in Appendix 1, and you can see how the first flip-flops are connected to the first fifo and the processing module.

This processing module is the one which need to be implemented now. As a blur processing uses only coefficients with value '1', simple adders will be enough to get the output.

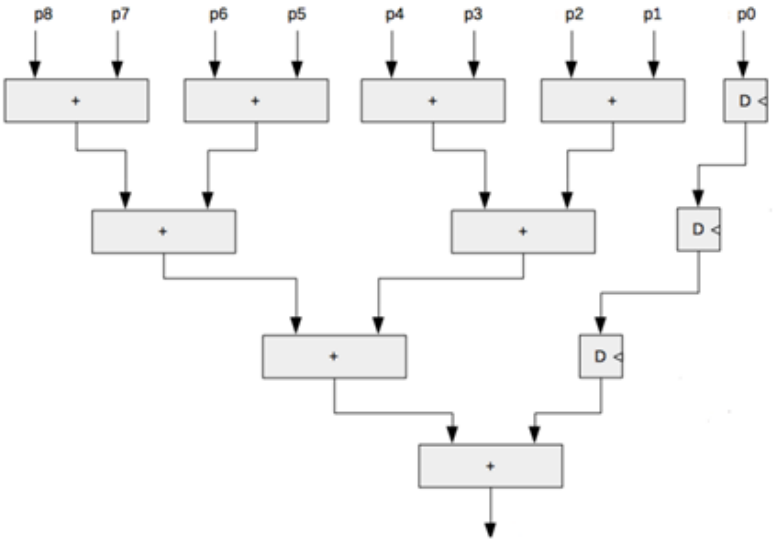

Figure 3.1: Framework of the blur processing

As their is 9 pixels to process, each pair of 2 pixels can be added together and a remaining 9^{th} pixel can be propagated into each layer of adders. At each layer we add together the results of the previous adders, and when it remains only one number, we add it to the 9^{th} pixel.

As the result of bit additions can lead to bigger memory size, the number of bits needs to be increased for coding the pixels. At least 1 bit per layer of adders has to be added to the encoding. Their are 4 layers of adders, so the pixels need 4 more bits. We increase their size from 8 bits to 12 bits.

For normalizing the result, it has to be divided by 9. A simple trick to operate a division with bit encoding is to operate a bit shift on the left. As the result is encoded on 12 bits, we can therefore keep its 10^{th} to 3^{rd} bits, what will do the same than shifting 3 times this bus on the left. Three left shifts has the same effect than a division per $2^3 = 8$. As we can only divide per 8, we can apply a mask to the pixels and count the neighbors of the actual pixel without its own value. The mask to apply is the following one : $\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$

Chapter 4

STATE MACHINE

A state machine can be used for a better management of the component. The following steps need to be operated one after each other :

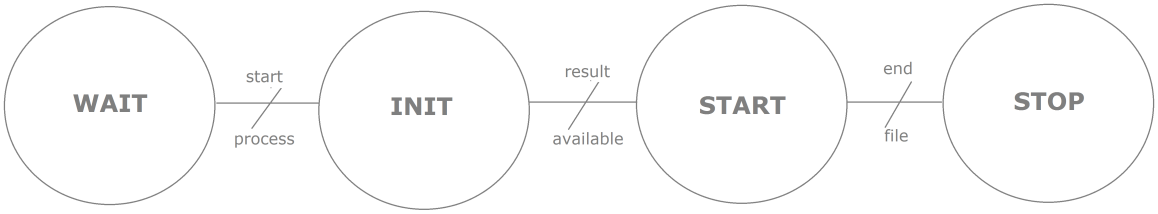

Figure 4.1: State machine for using the component

We first need to send a start signal to the component in order to tell it to start getting data and process them. This is the initialization step. During this step, we will feed the cache memory with pixels and the component has to receive at least nine pixels. In order to know if these pixels went through all the flip-flops, the *start_process* signal can be propagated to multiple flip-flops until it reaches an output *result_available* which can enable the next step of the state machine.

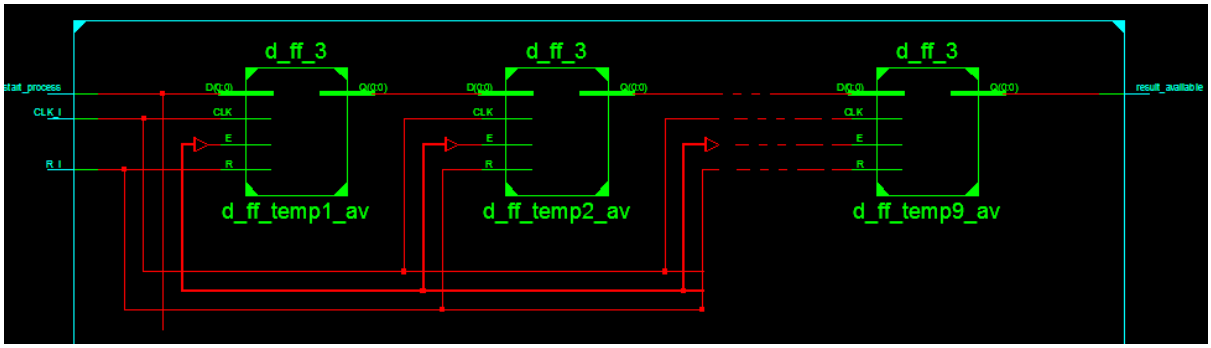

Figure 4.2: Flag signal propagated through flip-flops until the output

With the help of this flag , the user will know when does he have to start writing in the output file. Then, when the input file is totally read, the *start_process* signal is set to 0 and, as it takes time to propagate it, the component will finish processing the remaining pixels before setting *result_available* to 0.

Chapter 5

RESULT

The following results are the images obtained with different usages of the design explained in the previous chapters.

By not taking care of the initialization step, the component starts to output useless values. If you start to write in the output file directly from the initialization step, the resulting image will display a shift of 9 pixels on the right. The rows will actually not be split correctly. You can see in the following figure that the pixels which are suppose to be located on the right of the image are now on the left part of the image.

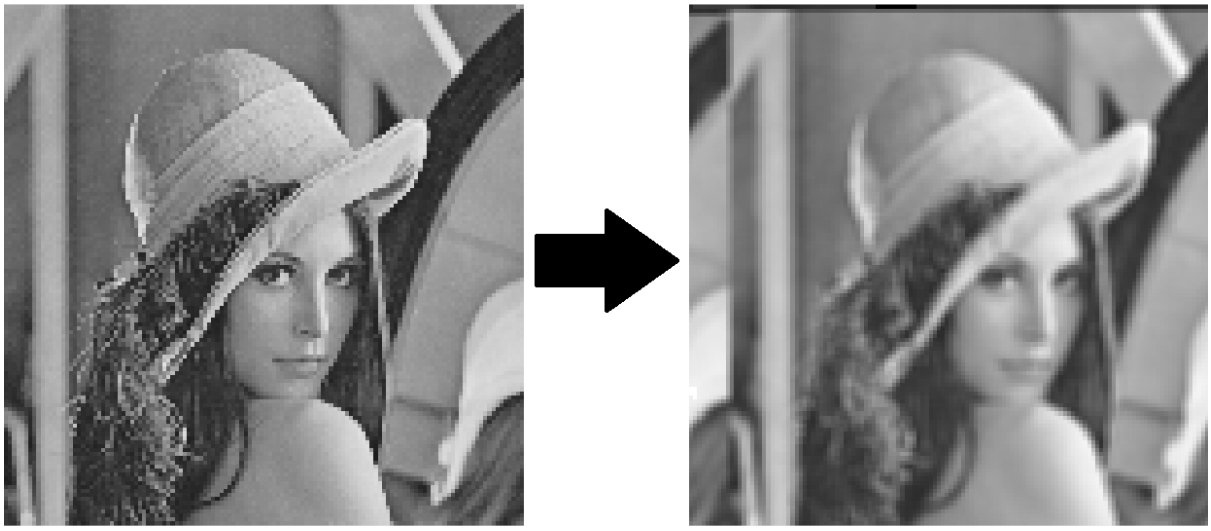

Figure 5.1: Smooth filter applied without initialization steps

Adding this initialization step, helps correcting it. We don't see any shift anymore, but a thick border of 2 pixels can be found on top of the image. That is because we still do not wait

long time enough to give the third row for the computation of the first pixel. We also have a border of 1 pixel in all the image because of some pixels computed from only 5 or 3 neighbors (in the borders or corners).

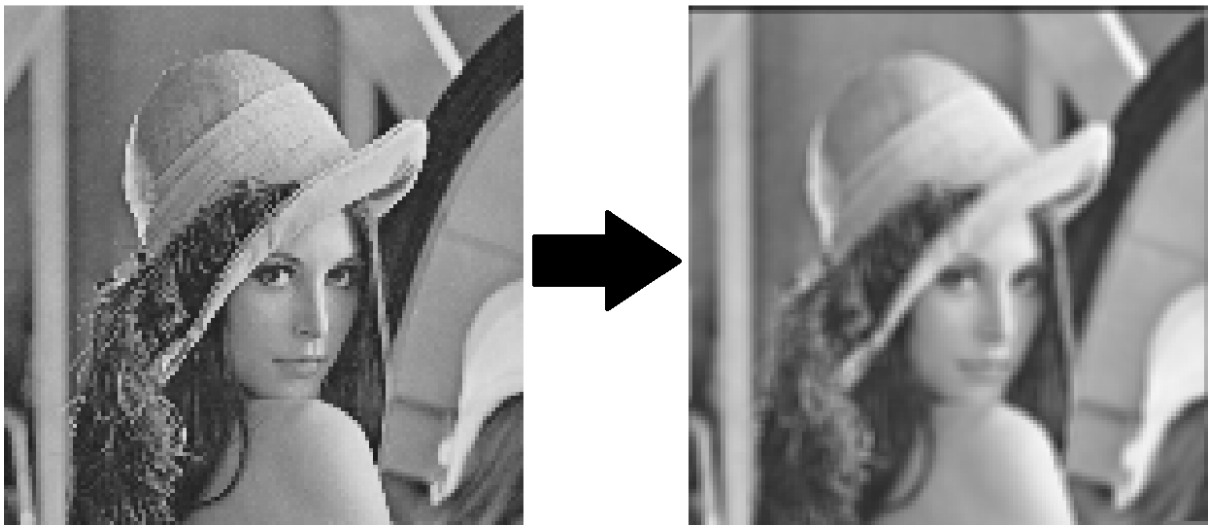

Figure 5.2: Smooth filter applied with undefined borders

In order to avoid this, we can wait longer before writing in the output file. For instance, waiting 128 clock cycles more will avoid having one dark row on top of the image. For the rest of the borders, we can fill them with black instead of undefined pixels by adding some steps in the state machine.

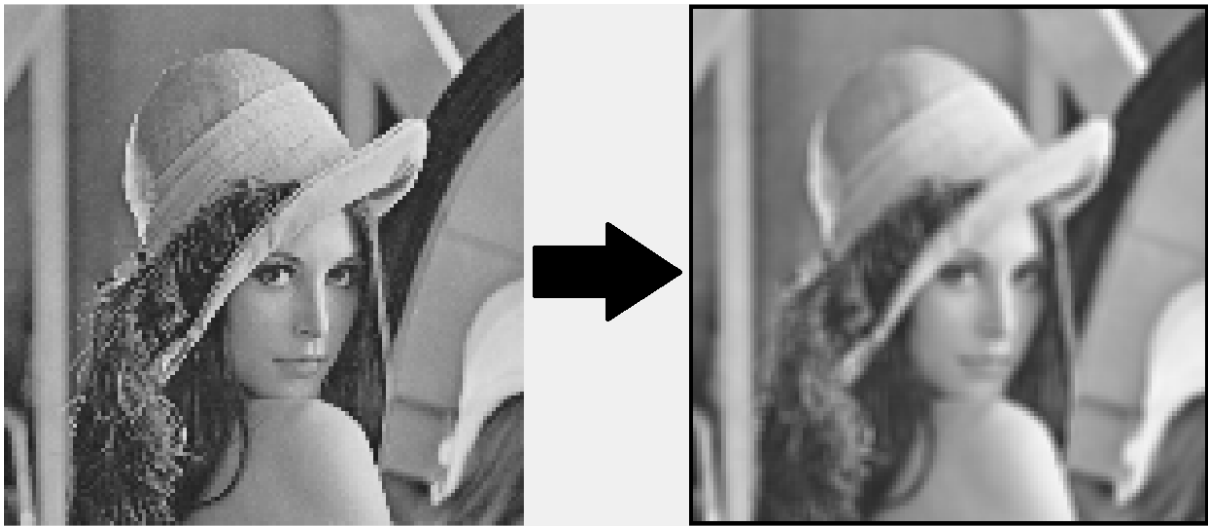

Figure 5.3: Smooth filter applied with zero padding

Chapter 6

Appendices

6.1 Appendix 1

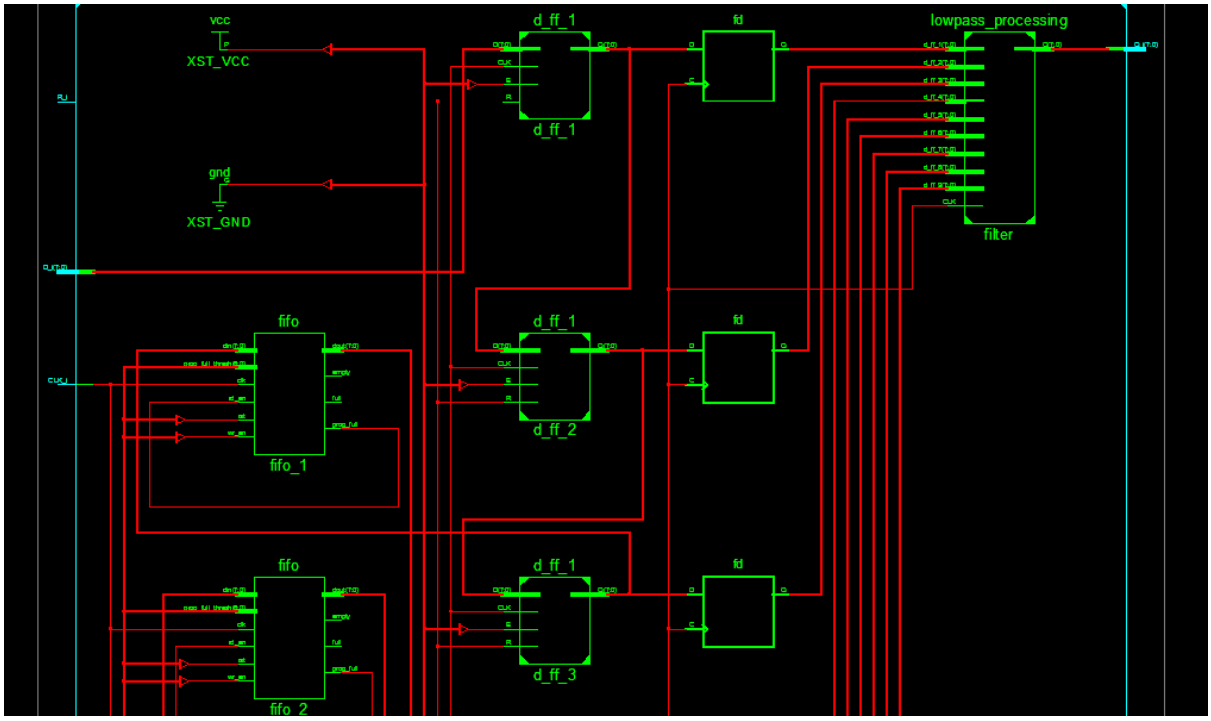

Figure 6.1: RTM schematic of the first part of the cache memory and the processing component

6.2 Appendix 2

Github Repository = https://github.com/yanik-porto/2DRealTimeFiltering.git