2D-Kernal Streaming, Low Pass Filter, Separable Convolution on an FPGA

Edward Lin

edwardclin2003@gmail.com

The objective of this project is to design an efficient, streaming, low pass filter FPGA module in Verilog HDL for a greyscale image of size at least 4 pixels in each dimension. The filter is a 3x3 kernel:

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Below is an outline of the steps taken to the complete the assignment. Following the outline is a more detailed explanation of each step, highlighting the thought process and decisions made which led to the final solution.

1. Examine the problem and algorithm for optimizations
2. Review target FPGA to understand resource constraints
3. Develop baseline architecture
4. Optimizations:
    a. Pipelining/Parallelism
    b. BRAM utilization
5. Performance evaluation
    a. Simulation results for functionality testing
    b. Throughput and latency
    c. Synthesis Results: ISE
6. Further improvements

1. Examine the problem and algorithm for optimizations

The project definition is to convolve a fixed separable 3x3 matrix filter across an image, where pixels are input serially in row major order and output serially as well. If more pixels are able to be passed in and out simultaneously, more architectural parallelism options are available to increase throughput.

The separable feature of the convolving filter allows for the reduction of computation for a pixel from $O(N^2)$ multiplications to $O(2N)$, where N=3 (the fixed filter height/width) in this assignment. However, instead of one filter pass, this approach requires two filter passes to achieve the same result, introducing the need for an intermediate buffer. A quick analysis shows that I/O-wise, the basic (non-separable) filter requires 9 memory reads for every pixel, while a separable requires 3 reads + 1 write + 3 reads. By separating the 2-D filter into 2 1-D filters, not only is computation saved, but also I/O is also reduced.

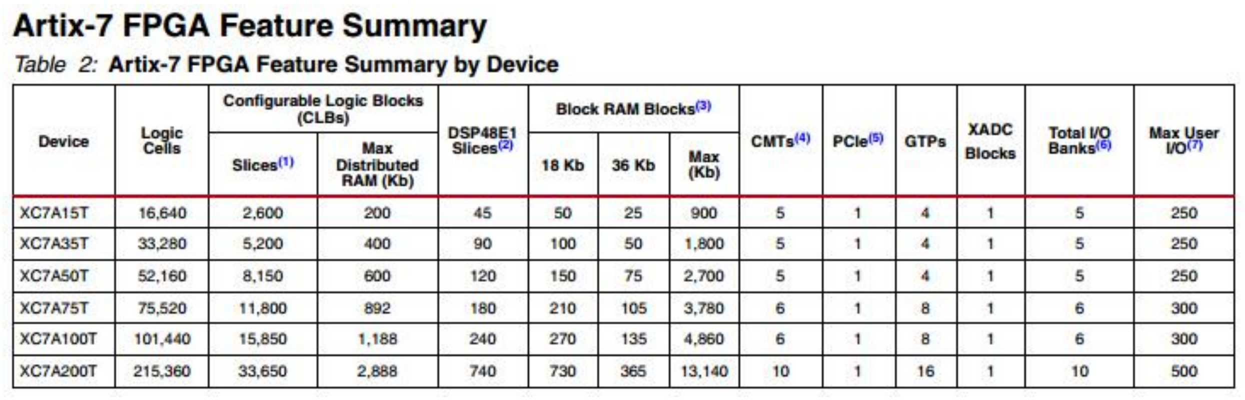
I looked into whether breaking down the 1-D filter could yield any further gains.

$$\frac{1}{4}[1\ 2\ 1] = \frac{1}{2}[1\ 1] * \frac{1}{2}[1\ 1]$$

The 1-D filter can be broken down to two filters. However, in this case breaking the filter into parts just increases computation requirements from 3 multiplies to 4. I/O also increases so this optimization does not yield benefit.

2. Review target FPGA to understand resource constraints

FPGA resources are limited. Before diving into an architecture for the assignment, I looked into one of the modern FPGAs mentioned in the assignment specification to understand most the most recent FPGA technology. Specifically, I focused in on the Xilinx Artix-7 since I'm more familiar with their family of FPGAs. As a low power, low cost FPGA, the need to optimize the design to fit the chip is critical. Xilinx's Virtex-7 family chips are roughly 40-100 times the smallest Artix-7 FPGA (Figure 1). BRAM and DSP slices are a scarce resource in the smaller chips.



Artix-7 FPGA Feature Summary

Table 2: **Artix-7 FPGA Feature Summary by Device**

Device	Logic Cells	Configurable Logic Blocks (CLBs)		DSP48E1 Slices[2]	Block RAM Blocks[3]			CMTs[4]	PCIe[5]	GTPs	XADC Blocks	Total I/O Banks[6]	Max User I/O[7]
		Slices[1]	Max Distributed RAM (Kb)		18 Kb	36 Kb	Max (Kb)						
XC7A15T	16,640	2,600	200	45	50	25	900	5	1	4	1	5	250
XC7A35T	33,280	5,200	400	90	100	50	1,800	5	1	4	1	5	250
XC7A50T	52,160	8,150	600	120	150	75	2,700	5	1	4	1	5	250
XC7A75T	75,520	11,800	892	180	210	105	3,780	6	1	8	1	6	300
XC7A100T	101,440	15,850	1,188	240	270	135	4,860	6	1	8	1	6	300
XC7A200T	215,360	33,650	2,888	740	730	365	13,140	10	1	16	1	10	500

Figure 1: Aurtix-7 Datasheet

Given a small FPGA like the Aurtix-7, we find that the power of 2 filter coefficients of the problem assignment advantageous in reducing the need for complicated multiplications which would likely require utilizing the DSP slices. However, the need for an intermediate buffer for the separated filters make BRAM resources a potential bottleneck if not designed properly.

3. Develop baseline architecture

Using the classic Lena image as an example, I illustrate the processing steps (Figure 2) that I started out with to come up with a foundation architecture which I could build upon. Then I discuss briefly the submodules in my architecture (Figure 3).

A. Processing steps

The first step in the module is to do the buffering of a row's pixels, given the instructions that the pixels will arrive in row order. Since area is an issue, using distributed RAM in this scenario would not yield good area utilization, so I expected to use a BRAM. The idea would be to hold just enough data in the BRAM that the vertical filter can begin processing. The reason to buffer here instead of between the vertical and horizontal filter is because each pixel is represented as 8 bits which means 2 consecutive pixels (16 bits) can be stored in a 18kB BRAM (0.5 of a 36kB BRAM) for an image width of 512 pixels. If buffering was done after filtering in a direction, the number of bits needed to be stored would require 1 whole 36kB BRAM to be used, which would be a waste of area.

Once a row of pixels are buffered, vertical y-axis filter computation can begin. Naturally processing the pixels in the order they arrive is desirable for streaming applications. Care must be made at edge boundaries to abide by the specification that entries outside the image are set to 0.

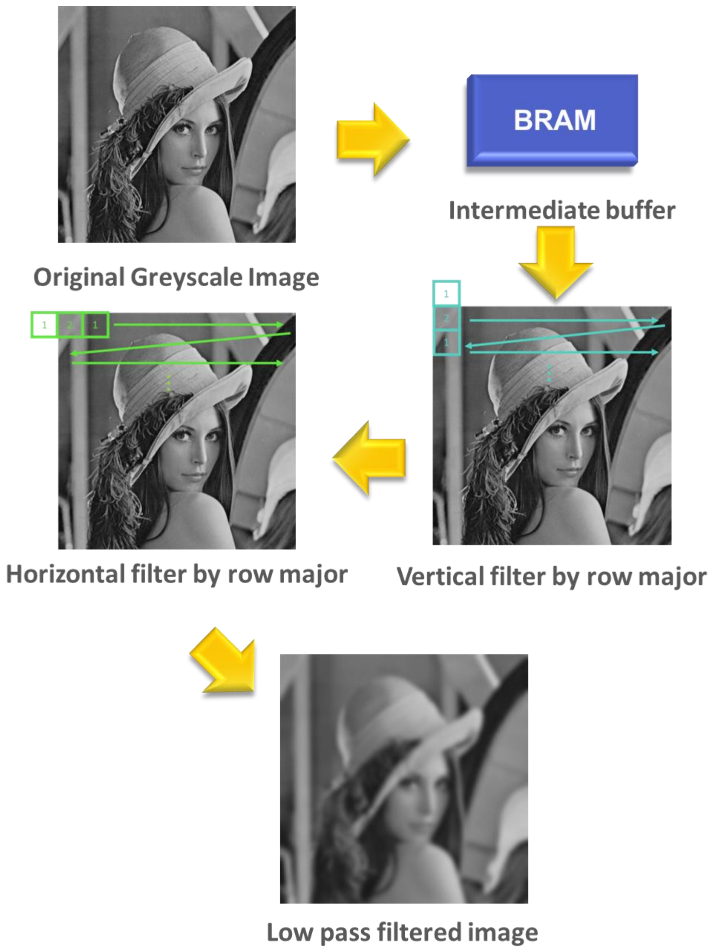


Figure 2: Process Steps

As vertical y-axis filtered pixels are generated, horizontal x-axis filter computation is done with a delay of 2 filtered pixels after the first row of pixels arrive. Once processed, the completely filtered pixels are sent out in a streaming fashion.

B. Architecture submodules

There are several submodules in my implementation (Figure 3). The first is handling the input handshaking to capture the streaming pixels. Next is the BRAM which hold 2 rows of pixels (2 row pixels are compacted into a single BRAM index to save space as seen in Figure 4). After a row of pixels have been buffered, the information is passed to the Vertical edge detector (isverticaledge.v) and Vertical y-axis filter (yfilter.v) which does the 1D filter. The output filtered pixel is fed into the Horizontal edge pixel detection (ishorizontaledge.v) and Horizontal x-axis filter (xfilter.v) once there are enough filtered pixels to begin processing. To support stalls, we utilize the a FIFO (hw_fifo.v) to hold pixels in the pipeline, while preventing new pixels from being fetched. Lastly, output handshaking is done to send the low pass filtered pixels back out.

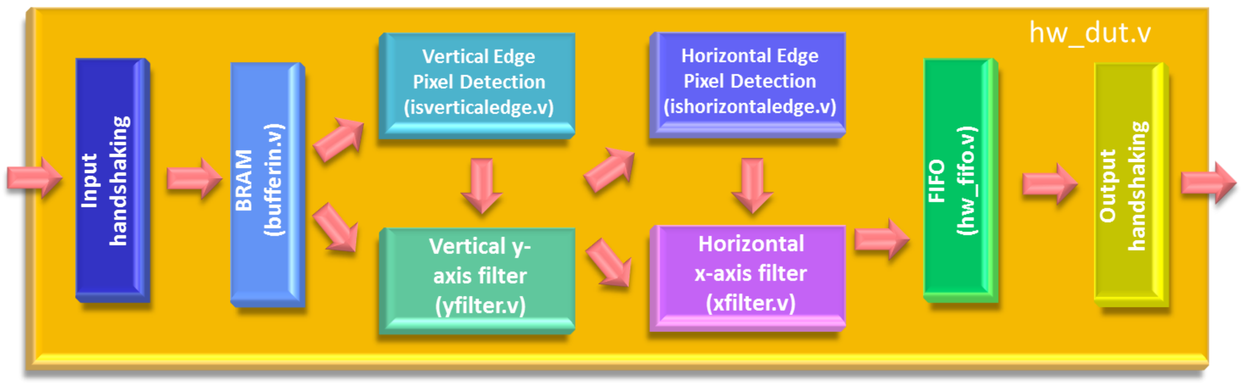

hw_dut.v
Input handshaking
BRAM (bufferin.v)
Vertical Edge Pixel Detection (isverticaledge.v)
Vertical y-axis filter (yfilter.v)
Horizontal Edge Pixel Detection (ishorizontaledge.v)
Horizontal x-axis filter (xfilter.v)
FIFO (hw_fifo.v)
Output handshaking

Figure 3: Block Architecture

4. Optimizations

In my design there are several optimizations I investigated to improve throughput, decrease area/BRAM, reduce timing delay, and overall latency. These optimizations has led to a streaming module with a latency of alittle more than 1 row worth of cycles, where each clock cycle will produce a pixel output, so long as there are no incoming stalls from signal px_out_ready. Below I discuss a few of these improvements.

A. Pipelining/Parallelism

My implementation is fully pipelined such that every cycle that a pixel is available on the input, an output pixel is produced. All the components are functioning in parallel to achieve maximum throughput.

Pixels are streamed in a BRAM (bufferin.v) through one port, while another port reads from the BRAM to retrieve the previous 2 rows' pixels for calculation of that pixel's Vertical filter value (yfilter.v). When there are 3 consecutive vertically filtered pixels ready (2 on a vertical edge boundary), the Horizontal x-axis filter (xfilter.v) can begin processing with throughput of 1 pixel per cycle, as the filter acts as a sliding window of 3 pixels.

The FIFO (hw_fifo.v) is used to buffer pixels that have been processed but cannot be consumed by the output module. If a stall command is given, the stall is sent to the input source to prevent sending more pixels that cannot adequately be handled. By buffering the pixels, when the output is ready to consume more pixels, the flow of 1 pixel/cycle can continue without interruption.

B. BRAM utilization and throughput

BRAM utilization is very important in this design because we are targeting a low power device. Initially the thought was that the design would require 3 separate BRAMs, one holding the pixels for a previous row, pixels for 2 rows back, and another to hold the new incoming pixel.

Since each BRAM is dual ported, I was able to reduce the BRAM utilization down to 1 BRAM (depending on size of an image row) by cleverly pipelining the data so only 2 rows of pixels are ever stored in BRAM.

Even though the Vertical filter requires 3 pixels for computation, only the most recent two needs to be stored in the BRAM at any point in time. The 3rd comes from the input stream and gets sent directly to the Vertical filter with the output from the BRAM. At the same time, the streamed input pixel and the previous row's pixel is written back to BRAM (fetched for the Vertical filter) on the other port (Figure 4). By doing so, each BRAM port is fully utilized and maintains the 1 cycle/pixel throughput. Because each pixel is 8 bits, the BRAM width is 16 bits. For images of a basic size of 512x512, the requirement would be 0.5 BRAM.

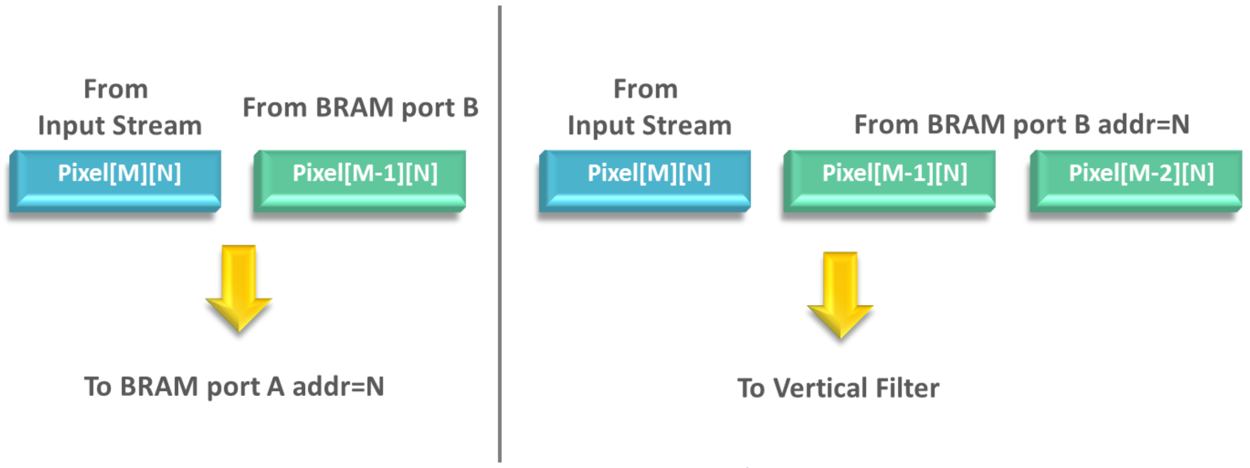


Figure 4: Dual Port BRAM: Write to port A/Read from port B

5. Performance evaluation

A. Simulation results for functionality testing

For simulation I used the hw_tbv script to evaluate correctness. The main issues of concern in a design like mine would be how stalls would affect processing since the design is fully pipelined. The output FIFO is large enough (16 deep) so when stalls occur it shouldn't overflow when stalls occur in a full pipeline. Traditionally I would write a python script then followed by a Verilog testbench to check correctness, but since a testbench was already provided I just used this for functionality testing.

B. Throughput and latency

Without stalls, the throughput of the design is 1 pixel/cycle. The latency in cycles is (# of pixels in 1 row) + 10 cycles. In the simulation script hw_tbv, test 1 had a row width of 19 and the overall cycles to return a result is 29 cycles.

C. Synthesis results

Using an older version of Xilinx ISE 14.7, I was able to get a rough estimate of timing and resource utilization of my design on a Aurtix-7 XCA100T, one of the larger chips. The minimum period is 2.718ns or a maximum frequency of 367.9 MHz. The resource utilization results are shown in Table 1.

Logic Utilization	Used	Available	Utilization
Slice Registers	226	126800	0%

Slice LUTs	192	63400	0%
Fully used LUT-FF pairs	111	421	26%
Bonded IOBs	44	210	20%
Number of Block RAM/FIFO	0.5	135	0%
BUFG/BUFGCTRLs	1	32	3%

Table 1: Logic utilization of design

6. Further improvements

For further improvements there's several directions I would look. The first is to look at timing to see if there's any obvious need/urge to improve it. Then I'd spend some time looking at each submodule within my design to see whether there were any more gates I could squeeze out.